POLITECNICO DI TORINO

Laurea Magistrale in Ingegneria del Cinema e dei Mezzi di Comunicazione

Politecnico di Torino

Tesi di Laurea Magistrale

Progettazione e generazione di un dataset sintetico fotorealistico orientato al grasp learning

Relatori

Prof. Andrea BOTTINO

Dr. Francesco STRADA

Candidato

Jacopo BERTOGLIO

Ottobre 2023

Abstract

L'identificazione della posizione e dell'orientamento che un braccio robotico deve assumere per ottenere una presa stabile di un determinato oggetto viene denominata *grasp synthesis*. Il problema in questione viene risolto, nell'ambito della *computer vision*, tramite dei dataset contenenti immagini che raffigurano oggetti di vario genere, utilizzati per allenare algoritmi orientati al *grasping*. Le immagini che compongono un dataset di questo tipo possono essere collezionate con apposite fotocamere nel mondo reale, comportando costi e tempistiche elevate, oppure in modo sintetico all'interno di un ambiente virtuale.

Il lavoro svolto per questa tesi prevede la costruzione di un dataset sintetico contenente immagini fotorealistiche, generate all'interno del *game engine* Unity tramite l'ausilio della High Definition Render Pipeline, una pipeline di render che permette di ottenere i risultati fotorealistici desiderati. Per avvicinarsi il più possibile alla realtà è stato sfruttato il motore fisico interno a Unity, effettuando la simulazione di un certo numero di oggetti che, cadendo, si dispongono su di una superficie piana in modo casuale. Modificando il tipo di illuminazione, l'ambiente di background, il numero di oggetti presenti nella scena e, di conseguenza, il livello di clutterizzazione degli stessi, si è andata ad impostare una difficoltà crescente all'interno del dataset.

La caratteristica principale del dataset costruito per questa tesi risiede nella segmentazione delle parti degli oggetti, la quale rende possibile il riconoscimento di singole parti di un oggetto orientate allo svolgimento di una specifica funzione, come ad esempio l'impugnatura di un coltello o il tappo di una bottiglia. La segmentazione per parti viene resa disponibile sotto forma di particolari immagini, definite maschere, che associano ciascuna parte di ogni oggetto ad un unico colore, fornendo l'associazione colore-parte tramite un apposito dizionario.

Per garantire il raggiungimento di risultati soddisfacenti è necessario che la scala del dataset sia opportunamente grande per poter allenare a dovere l'algoritmo di *grasping*. Per questo motivo, il dataset è composto in totale da 174.000 immagini e 88 oggetti, appartenenti a 38 differenti categorie. Il dataset è suddiviso in una porzione più consistente dedicata a *train* e *validation* e in una porzione più ridotta dedicata al test.

Indice

Elenco delle tabelle

Elenco delle figure

Acronimi

URP
Universal Render Pipeline

HDRP
High Definition Render Pipeline

API
Application Programming Interface

HDR
High Dynamic Range

SOLO
Synthetic Optimized Labeled Objects

MTL
Material Template Library

FOV
Field Of View

Capitolo 1

Introduzione

L'oggetto del lavoro svolto per la presente tesi consiste nella creazione di un dataset sintetico fotorealistico, su richiesta del laboratorio di ricerca VANDAL (Visual And Multimodal Applied Learning Laboratory) del Politecnico di Torino.
Il dataset in questione è orientato al supporto dell'addestramento di un algoritmo di *grasp learning*, che dovrà analizzare sia le immagini sia le informazioni contenute nel dataset al fine di riconoscere oggetti specifici e di determinare le pose di *grasp* ad essi associate. Le pose di *grasp* rappresentano le diverse modalità con cui un braccio robotico può afferrare un oggetto in modo stabile dal piano posto di fronte ad esso.
Le immagini incluse nel dataset verranno catturate all'interno di un ambiente virtuale utilizzando il *game engine* Unity, in particolare implementando le funzionalità offerte dal Perception Package, un *package* pensato per la generazione di dataset sintetici di grandi dimensioni nell'ambito della *computer vision* [1]. Queste immagini rappresenteranno una varietà di oggetti casalinghi e di uso comune, disposti casualmente e in modo disordinato su una superficie di appoggio.
All'interno dei vari scenari di generazione saranno inclusi un numero variabile di oggetti, posizionati a diverse distanze tra loro e con livelli variabili di sovrapposizione. Questo sistema è stato adottato al fine di introdurre un fattore di complessità crescente all'interno del dataset, permettendo un'ampia variazione di situazioni e casistiche per la futura applicazione dell'algoritmo di *grasp learning*.
Un altro aspetto implementato per aumentare la complessità degli scenari è l'uso di un tavolo come superficie di supporto, che agirà come elemento di disturbo, associato ad un sistema di illuminazione non uniforme che varierà nel corso della generazione del dataset.
Un elemento fondamentale che distingue questo dataset da altri presenti in letteratura è la presenza della segmentazione delle parti degli oggetti. Questa caratteristica consente all'algoritmo di *grasp learning* di identificare le diverse parti che costituiscono un oggetto. Tali parti, se rappresentano un punto di presa, offrono funzionalità

diverse per uno stesso oggetto.
I dataset di immagini orientati al *grasp learning* che sono stati analizzati e confrontati con il dataset generato per la presente tesi sono approfonditi nel capitolo 2, e sono i seguenti:

- GraspNet-1Billion [2];
- ACRONYM [3];
- TaskGrasp [4].

Questi dataset sono stati utilizzati come punto di riferimento per stabilire la scala del dataset in fase di pianificazione. Il dataset comprenderà un totale di 174.000 immagini, attingendo da un insieme di 114 oggetti appartenenti a 38 categorie differenti.
La scelta di generare le immagini all'interno di un ambiente virtuale, ma in modo fotorealistico, è stata presa per sfruttare i vantaggi derivanti dalla flessibilità e dalla velocità della generazione in un ambiente virtuale, combinati con la fedeltà al mondo reale garantita da un rendering fotorealistico. Questo approccio permette di ottenere immagini sintetiche che possono essere trasferite ed utilizzate in contesti reali.

La struttura della tesi è organizzata in sette capitoli. Nel capitolo 2, verrà fornita una panoramica sulla *grasp synthesis*, con un'analisi dei dataset più significativi in questo ambito. Nel capitolo 3, saranno elencati e presentati i software necessari per la generazione del dataset, nonché il processo di raccolta e adattamento dei modelli 3D corrispondenti agli oggetti contenuti nel dataset.
Il capitolo 4 descriverà le specifiche tecniche del dataset, comprese la struttura delle immagini e dei file contenuti, oltre alla differenziazione degli scenari di simulazione.
Nel capitolo 5, verrà esaminata la metodologia utilizzata per la raccolta e la categorizzazione gli oggetti, inclusa la segmentazione delle loro parti e l'importazione degli asset in Unity.
Il capitolo 6 si concentrerà sulla gestione delle simulazioni effettuate in Unity, sulla generazione degli oggetti e loro gestione tramite simulazione fisica, e sull'utilizzo e la personalizzazione del Perception Package per generare il dataset secondo le specifiche richieste. Nel capitolo 7, saranno presentate le statistiche complete del dataset, inclusi i tempi di generazione e la quantità di oggetti istanziati, in relazione alle diverse sezioni che compongono il dataset.

Capitolo 2

Stato dell'Arte

2.1 Introduzione al grasping ed al deep learning per la grasp synthesis

Il termine *grasping*, utilizzato nell'ambito della robotica, si riferisce all'atto eseguito da un braccio meccanico nel prendere un oggetto posto su una superficie. Questa azione, sebbene complessa, riveste un ruolo fondamentale poiché costituisce la base per la realizzazione di azioni e interazioni più articolate [5].
Il problema che si incontra nel raggiungere questo obiettivo è noto come *grasp synthesis*, il quale consiste nel determinare una serie di pose di *grasp*, parametrizzate da coppie di posizione e rotazione del *gripper* del braccio meccanico, che soddisfino una predeterminata serie di criteri relativi al compito da svolgere [6]. Inoltre, tali pose di *grasp* devono garantire una presa stabile sull'oggetto. Solitamente il *gripper* è rappresentato in maniera semplificata da un meccanismo a pinza, composto da due dita che si avvicinano l'un l'altra per stringere la parte dell'oggetto da afferrare.
Come sottolineato da Newbury et al. in [7], l'approccio maggiormente utilizzato in robotica per sintetizzare le pose di *grasp* è quello di applicare il *deep learning* alle immagini. Questo metodo, accompagnato da dataset di grandi dimensioni, permette di ottenere risultati robusti e precisi, a differenza di metodi analitici o di *machine learning* tradizionali, più popolari in passato. L'utilizzo e lo sviluppo del *deep learning* applicato alle immagini consentono di esplorare funzionalità più avanzate, come il riconoscimento di oggetti in ambienti disordinati e le relazioni tra essi, la comprensione del linguaggio naturale e la sua traduzione in azioni robotiche, oltre all'apprendimento *end-to-end*.
I metodi *data-driven* per la *grasp synthesis* sono composti da dati che possono avere diverse origini, elencate da Newbury et al. in [7]: nuvole di punti, *voxel grid*,

immagini RGB-D[1] ed immagini di *depth*. Le nuvole di punti sono una struttura geometrica per la rappresentazione di geometrie 3D e, rispetto alla rappresentazione tramite *mesh* 3D, offrono complessità minore e flessibilità maggiore. Le nuvole di punti sono la tipologia di dati più utilizzata per i metodi di *deep learning*, grazie anche alla diffusione di *network* come PointNet [8], in grado di interpretare direttamente le nuvole di punti come dati in input.

2.2 Esempi di dataset per l'addestramento al grasping

La generazione di dataset volti ad allenare algoritmi di *grasping* è un problema che può essere affrontato in diversi modi, variando tecniche e tecnologie.
Tra gli aspetti che contraddistinguono i differenti dataset sono presenti: la modalità di acquisizione dei dati, la rappresentazione delle pose di *grasp* e le metriche di valutazione delle stesse, la variabilità delle scene e il numero di oggetti presenti.
Alcuni dataset, cronologicamente precedenti a quelli che si andranno ad analizzare, implementano annotazioni manuali delle pose di *grasp*, portando il risultato ad essere inevitabilmente ridotto nella scala e non esaustivo.
L'acquisizione di dati in modo sintetico in ambienti virtuali aumenta la scala del dataset, ma bisogna tenere conto del disallineamento con la realtà per le applicazioni che intendono usare i dati acquisiti in implementazioni nel mondo reale.
Di seguito si andranno ad esaminare tre differenti approcci, ovvero i dataset GraspNet-1Billion, ACRONYM e TaskGrasp.

2.2.1 GraspNet-1Billion

Il dataset GraspNet-1Billion [2] offre un'ampia raccolta di pose di *grasp* a 6 gradi di libertà, associate ad oggetti casalinghi e di uso comune.
La caratteristica peculiare di questo dataset, e ciò da cui prende il nome, è l'estrema quantità di pose di *grasp* fornite al suo interno: oltre il miliardo. Il *network* GraspNet si basa su una pipeline automatizzata in due step, la quale genera le annotazioni relative alle pose di *grasp* per ogni scena. Il dataset contiene, nella sua interezza, 97.280 immagini RGB-D raffiguranti 190 scene differenti e, per ogni scena, le pose di *grasp* annotate variano tra 3 milioni e 9 milioni.
Le immagini sono state catturate dalle camere RGB-D Intel RealSense 435 e Kinect 4 Azure, tramite un braccio robotico che, ripetendo una traiettoria predefinita, si spostava lungo 256 punti su un quarto di sfera. Ogni scena è composta da 10

[1]Il canale D si riferisce alle informazioni di profondità, ovvero traduce in un valore su scala di grigi la distanza tra un punto nell'immagine e il sensore.

oggetti disposti in modo casuale e disordinato su una superficie piana. Gli oggetti in questione sono in totale 88 e ognuno è provvisto di una corrispettiva *mesh* 3D ad alta risoluzione. L'origine degli oggetti è varia: 32 sono stati presi dal dataset YCB [9], 13 da DexNet 2.0 [10] e 43 sono oggetti di proprietà aggiunti al dataset.

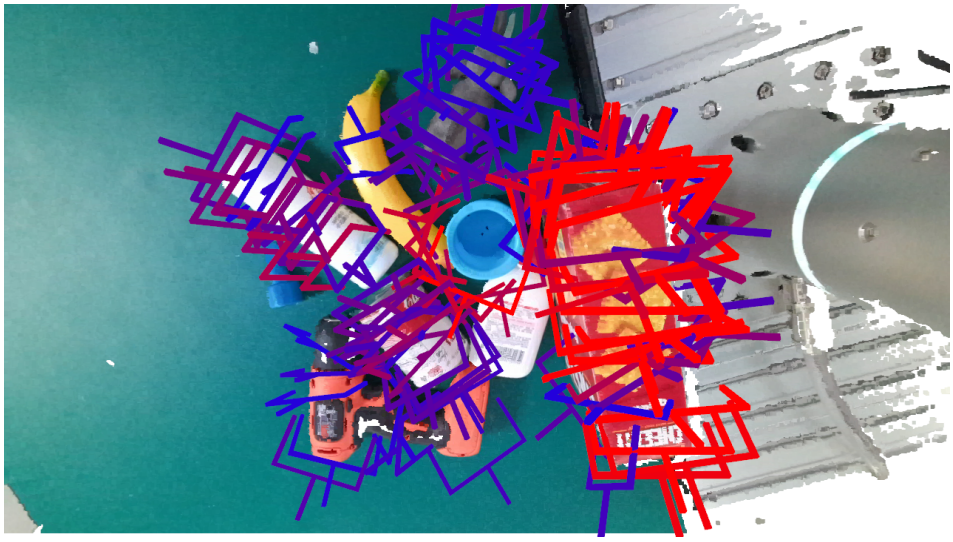


Figura 2.1: Esempio di output di GraspNet-1Billion con le annotazioni delle pose di *grasp* [2].

Delle 190 scene che compongono il dataset, 100 sono state usate per il *training* e 90 per il *testing*. Le scene di *testing* sono state ulteriormente suddivise in 3 categorie: 30 scene con oggetti conosciuti, 30 scene con oggetti sconosciuti ma simili ad oggetti visti e 30 scene con oggetti sconosciuti. Esperimenti di robotica condotti nel mondo reale hanno dimostrato l'efficacia del *network* GraspNet, in grado di fornire risultati sovrapponibili con la realtà.

2.2.2 ACRONYM

Il dataset ACRONYM [3] si basa su una simulazione fisica per la generazione di pose di *grasp*.
Gli oggetti a cui fa riferimento sono in totale 8872, appartenenti a 262 categorie diverse. Le *mesh* 3D degli oggetti provengono da ShapeNetSem [11], un dataset di modelli 3D di oggetti casalinghi dotati di informazioni intrinseche sulle proprietà fisiche. Ogni oggetto è provvisto di diverse pose di *grasp*, applicate e valutate tramite FleX [12], un simulatore fisico particellare di proprietà di Nvidia, che permette di rendere i risultati ottenuti più vicini al mondo reale. In totale all'interno del dataset sono presenti 17,744 milioni di annotazioni di pose di *grasp*. In aggiunta alle scene contenenti un singolo oggetto, il dataset è inoltre provvisto di un meccanismo

procedurale per generare scene composte da più oggetti disposti in modo disordinato su di una superficie piana.

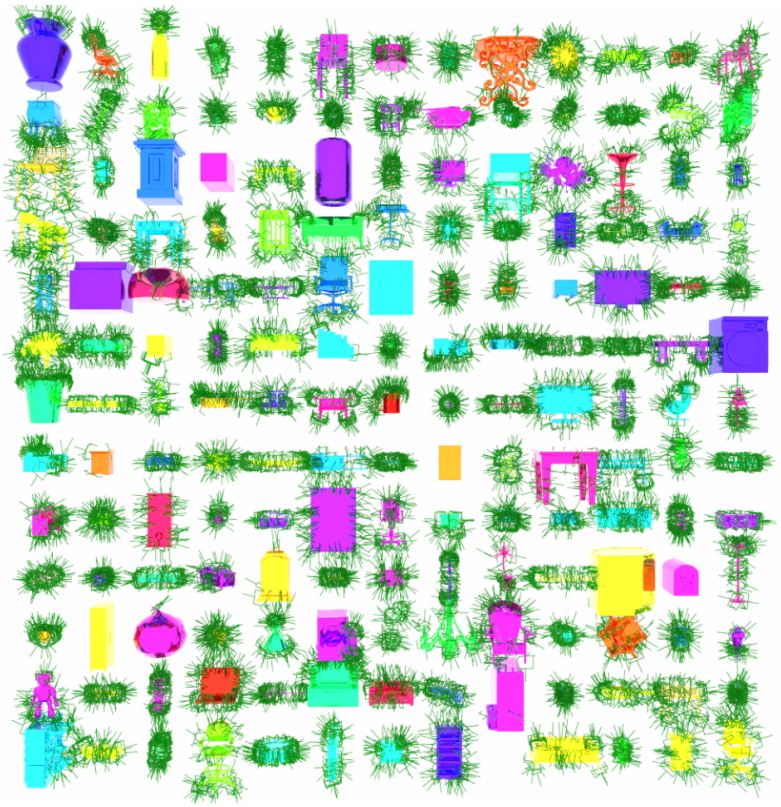


Figura 2.2: Annotazioni di pose di *grasp* presenti in ACRONYM [3].

L'output del dataset non è composto da immagini, bensì da codice basato su pyrender[2] per renderizzare immagini di *depth*, maschere di segmentazione e nuvole di punti.

2.2.3 TaskGrasp

Il dataset TaskGrasp [4] fornisce un'ampia quantità di *grasp* orientati a specifiche task, avendo come obiettivo quello di colmare la differenza nel modo in cui gli umani e i robot raccolgono gli oggetti. Uno specifico oggetto, infatti, verrà preso in modi diversi a seconda dell'utilizzo che se ne vuole fare.
Il numero di oggetti presenti nel dataset è pari a 191, provenienti da 75 diverse classi nominate secondo le convenzioni del database lessicale WordNet [13], mentre

[2]Libreria Python per il rendering basato sulla fisica e la visualizzazione.

le task associate agli oggetti sono pari a 56.
Sono state ricavate le nuvole di punti RGB-D per ciascun oggetto, utilizzando una camera Realsense D415 montata su un braccio robotico LoCoBot per eseguire le scansioni 3D.
In seguito, sono state campionate 600 pose di *grasp* stabili per ogni oggetto, prendendone poi 25 come pose rappresentative, che permettono di garantire una copertura completa dell'oggetto.

Figura 2.3: Alcuni oggetti presenti in TaskGrasp con l'annotazione delle pose di *grasp* in relazione ad una specifica task [4].

Capitolo 3

Software utilizzati

3.1 Unity

Unity[1] è una piattaforma di sviluppo 3D e *game engine* di proprietà di Unity Technologies, ampiamente utilizzato nell'industria per sviluppare applicazioni interattive (tipicamente videogiochi) 3D e 2D, eseguibili su desktop, mobile, web, realtà aumentata e realtà virtuale. È disponibile per i sistemi operativi Windows, MacOs e Linux ed è gratuito nella sua versione base.
Il linguaggio di programmazione implementato in Unity per gestire e modificare le componenti di un'applicazione via script è il C#.
Unity mette a disposizione per i suoi utenti lo Unity Asset Store[2], da cui è possibile acquistare o scaricare gratuitamente asset 3D e 2D, tool di sviluppo, librerie audio, *add-on* e tutto ciò che può servire per sviluppare la propria applicazione.
Creando un nuovo progetto in Unity è possibile scegliere la tipologia di pipeline di render da implementare, in base al look visivo che si vuole ottenere, alle specifiche tecniche e grafiche e alla piattaforma di destinazione.
È possibile creare tramite script una propria pipeline di render personalizzata, ma Unity ne offre tre già pronte all'uso:

- Built-in Render Pipeline: la pipeline implementata di default da Unity. Offre scarse possibilità di personalizzazione;

- Universal Render Pipeline (URP): è pensata per essere facilmente personalizzabile tramite script e per adattare le proprie impostazioni grafiche a diverse piattaforme;

[1] https://unity.com/

[2] https://assetstore.unity.com/

- High Definition Render Pipeline (HDRP): anch'essa personalizzabile tramite script. Offre i migliori risultati a livello grafico ed è ottimizzata per le piattaforme di ultima generazione.

Nell'ambito di questo lavoro di tesi, per la creazione del dataset sintetico, è stata scelta l'HDRP, l'unica pipeline di render in grado di fornire il risultato fotorealistico desiderato. Questa pipeline implementa l'utilizzo del *ray tracing*, il quale gestisce la luce e le sue interazioni con gli oggetti basandosi su leggi fisiche e ottiche.

3.1.1 Perception Package

Durante la fase di generazione del dataset, è stato fondamentale il Perception Package [1] [14], un *toolkit* open source altamente personalizzabile, sviluppato da Unity Technologies, che permette la creazione di dataset sintetici per la *computer vision*.
La caratteristica principale di questo package consiste nella possibilità di randomizzare specifici parametri, in modo da ottenere un dataset il più vario e completo possibile, andando a generare scenari sempre differenti.
Il Perception Package comprende dei componenti, denominati Labeler, che permettono, per ogni frame, di catturare informazioni di *ground truth*, come *bounding box* 2D, *bounding box* 3D, *instance segmentation* e *semantic segmentation*. Viene inoltre offerta la possibilità di creare tramite script dei Labeler personalizzati, per ricavare informazioni specifiche dalle immagini catturate.

3.2 Blender

Blender[3] è un software gratuito e open source, orientato allo sviluppo 3D in tutte le sue sfaccettature. Blender offre infatti strumenti di modellazione, *rigging*, animazione, *texturing*, *shading*, *rendering*, ecc.
La sua caratteristica di essere open source e completamente modificabile tramite API Python, ha permesso la nascita di una community molto attiva, con una conseguente ampia offerta di *add-on* per qualunque esigenza, gratuiti e a pagamento. Con il termine *add-on* si intende uno script Python che estende le funzionalità di Blender: alcuni sono già integrati nel software e disabilitati di default, altri possono essere installati tramite un file .zip.
In alcune schermate del programma è possibile utilizzare un'interfaccia a nodi, combinando singoli nodi per ottenere il risultato desiderato. In particolare è possibile interagire con l'editor a nodi nella creazione di *shader* personalizzati, nella creazione di *Geometry nodes* e nel *compositing*.

[3] https://www.blender.org/

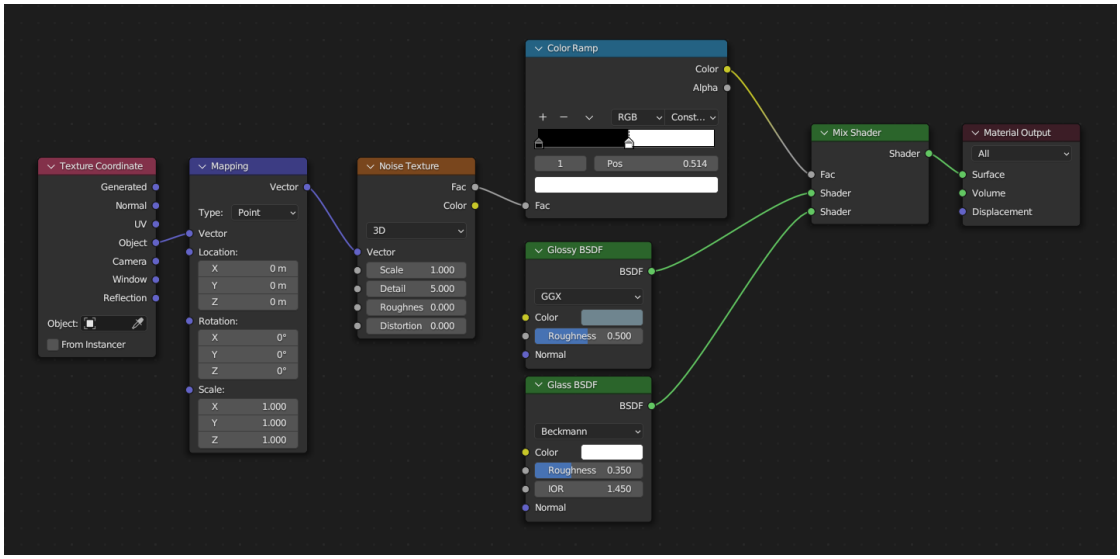


Figura 3.1: Esempio di combinazione di nodi per la creazione di uno *shader*.

Il processo di rendering, ovvero il passaggio dall'ambiente 3D ad un'immagine 2D, può essere effettuato all'interno di Blender tramite tre differenti motori di *rendering*:

- Workbench: utilizzato per ricavare preview e durante il processo di modellazione ed elaborazione degli elementi 3D;
- Eevee: motore di rendering real time basato sulla fisica;
- Cycles: motore di rendering fotorealistico dotato di *path tracer* basato sulla fisica.

3.3 Substance 3D Painter

Substance 3D Painter è un software di proprietà di Adobe[4], dedicato al texturing fotorealistico di modelli 3D. Fa parte del pacchetto Substance 3D Collection, un insieme di programmi dedicati a diversi aspetti della pipeline di produzione 3D, tra cui la modellazione, il *texturing*, la creazione di materiali procedurali e la composizione di scene.
Substance 3D Painter mette a disposizione degli utenti una vasta scelta di materiali di base da applicare ai propri modelli 3D, spaziando tra differenti categorie: materiali

[4]https://www.adobe.com/

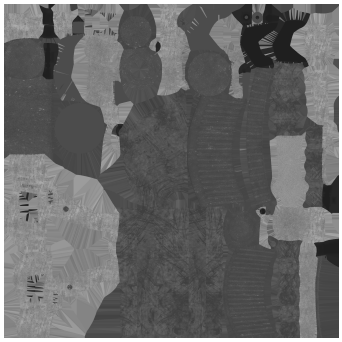
organici, metallici, fibre tessili, legnosi, plastici, ecc. Diversi materiali possono essere combinati tra loro tramite un sistema a livelli, andando a creare materiali più complessi e personalizzati.
La caratteristica principale di questo programma è però quella di poter dipingere direttamente sul modello 3D, come si potrebbe fare nella realtà, scegliendo tra diverse tipologie di pennelli ed effetti, in modo da rendere il più realistico possibile il risultato finale, andando ad aggiungere dettagli in modo estremamente preciso.
I materiali creati su misura per i propri modelli possono essere infine esportati sotto forma di texture, ovvero un set di immagini a colori o in scala di grigi che, applicate in combinazione tra loro, determinano in che modo la superficie dell'oggetto reagisce all'interazione con la luce. Le immagini di texture maggiormente utilizzate sono:

- *base color*: corrisponde all'informazione riguardante il colore di base della superficie;
- *roughness*: indica se la superficie è rugosa o liscia, con un risultato rispettivamente opaco o lucido. La texture è in scala di grigi: il bianco indica il valore massimo di rugosità, il nero il valore minimo;
- *metallic*: indica se la superficie è metallica, anche questa texture è in scala di grigi. Tipicamente si utilizzano solo valori bianchi (superficie metallica) o neri (superficie non metallica) e non valori intermedi;
- *normal*: codifica le informazioni di dettaglio superficiale tridimensionale, associando ai canali R,G, e B le direzioni X, Y, e Z delle normali nello spazio. È utile in quanto permette di avere dei dettagli superficiali precisi senza che la *mesh* dell'oggetto sia necessariamente ad alta risoluzione;
- *displacement*: definisce l'altezza di ogni punto della mesh, generando sporgenze e rientranze sulla superficie. A differenza della *normal*, è necessario che la mesh abbia una risoluzione sufficientemente alta per generare dei dettagli precisi.

3.4 Substance 3D Sampler

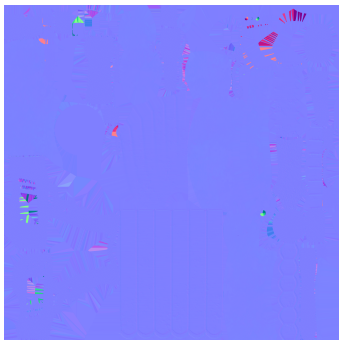
Substance 3D Sampler è, allo stesso modo di Substance 3D Painter, integrato nel pacchetto di Adobe Substance 3D Collection. Questo software è pensato per digitalizzare immagini provenienti dal mondo reale, trasformandole in immagini di texture, *mesh* 3D o ambienti HDR.
Con ambienti HDR si intendono immagini a 360 gradi che, applicate all'interno di un ambiente 3D, forniscono un'illuminazione globale alla scena che coincide con quella dell'ambiente reale dell'immagine.

(a) *Base color*



(b) *Roughness*

(c) *Metallic*



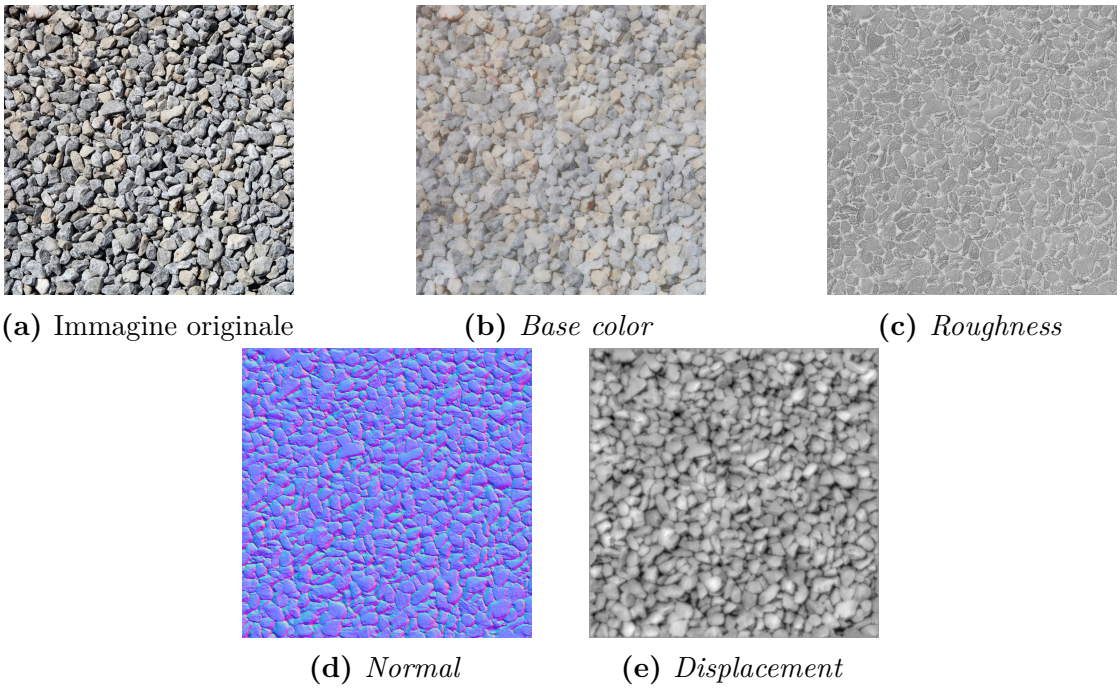
(d) *Normal*

Figura 3.2: Esempio di immagini di texture per uno stesso modello 3D.

Figura 3.3: Render di modello 3D con immagini di texture mostrate in figura 3.2 applicate all'interno di Blender.

La funzionalità utilizzata nell'ambito di questa tesi è Image to Material, la quale sfrutta il *machine learning* per generare delle texture in output avendo una singola

immagine in input. Viene sfruttato il *machine learning* per la generazione di *normal map* precise e dettagliate, andando a riconoscere oggetti e forme all'interno dell'immagine permette, inoltre, di rimuovere le ombre presenti originariamente nell'immagine dal *base color*. Al momento questa funzionalità è disponibile solo su sistemi operativi Windows e Linux associati ad una scheda grafica Nvidia.



(a) Immagine originale **(b)** *Base color* **(c)** *Roughness*

(d) *Normal* **(e)** *Displacement*

Figura 3.4: Esempio di immagini di texture generate da Substance 3D Sampler a partire da una fotografia.

Figura 3.5: Texture generata in figura 3.4 applicata all'interno di una scena di Blender.

Capitolo 4

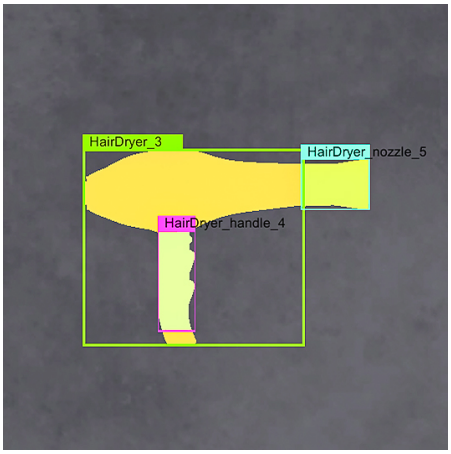
Specifiche del dataset

Il dataset creato nell'ambito di questa tesi, generato all'interno di Unity tramite il Perception Package, ha come scopo quello di rappresentare ed estrarre informazioni da diverse scene virtuali fotorealistiche, caratterizzate da una disposizione casuale di oggetti su una superficie piana. Il dataset sarà orientato al *grasping*, dovrà dunque offrire la possibilità di riconoscere gli oggetti contenuti nelle scene, applicando ad essi una serie di possibili pose di *grasp*, per allenare tramite *deep learning* un algoritmo che dovrà replicare queste azioni nel mondo reale.
L'aspetto principale su cui si concentra questo dataset, per il quale offre un aspetto aggiuntivo rispetto ai dataset più diffusi attualmente, è la segmentazione delle parti degli oggetti. La suddivisione di un oggetto nelle sue diverse parti permette di differenziare l'azione da svolgere in base a dove viene effettuato il *grasping*. Ad esempio, una padella comprende la maniglia, una bottiglia il tappo, un trapano l'impugnatura, ecc.
La maggior parte degli oggetti presenti all'interno del dataset offrono la suddivisione in parti, adeguatamente contrassegnate da apposite *label*, da cui poter essere afferrati e manipolati.

4.1 Struttura

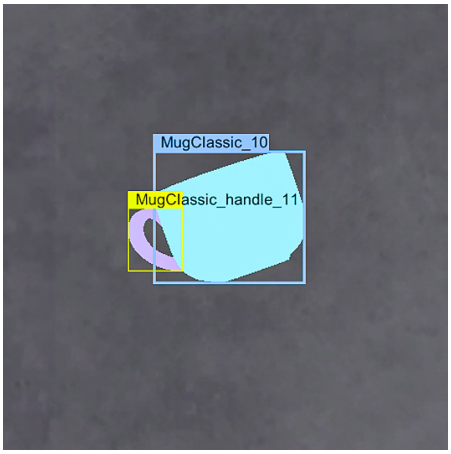
Il dataset è stato costruito tramite una ripetuta generazione di una serie di oggetti all'interno di un ambiente, lasciati cadere da una certa altezza e variando le impostazioni di generazione per ciascuna sezione di cattura dei dati.
Per ogni generazione casuale di oggetti, denominata "iterazione", una camera virtuale effettua 5 catture da angolazioni diverse, spostandosi lungo una traiettoria circolare.

(a) Oggetto *HairDryer*



(b) Parti dell'oggetto *HairDryer*

(c) Oggetto *MugClassic*



(d) Parti dell'oggetto *MugClassic*

Figura 4.1: Oggetti affiancati alle proprie maschere di segmentazione delle parti.

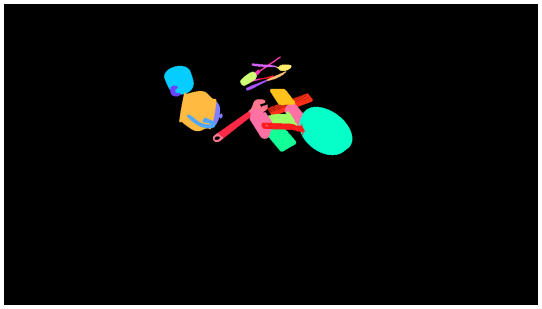
Ognuno dei 5 frame catturati sarà riportato in output sotto forma di 4 immagini differenti:

- immagine RGB;
- maschera di segmentazione delle parti degli oggetti;
- maschera di segmentazione degli oggetti;
- immagine di *depth*.

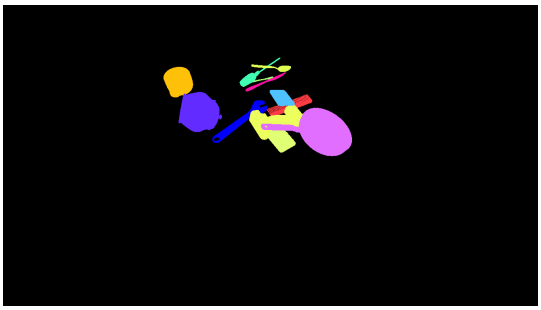
Ciascuna immagine è in formato .png, con una profondità di 32 bit ed una risoluzione di 1280 x 720 pixel.
Per incorporare informazioni aggiuntive all'interno del dataset riguardo la scena

e gli oggetti in essa presenti, è stato utilizzato uno schema di output fornito dal Perception Package, denominato SOLO (Synthetic Optimized Labeled Objects) [15], il quale inoltre genera in output le immagini precedentemente citate.

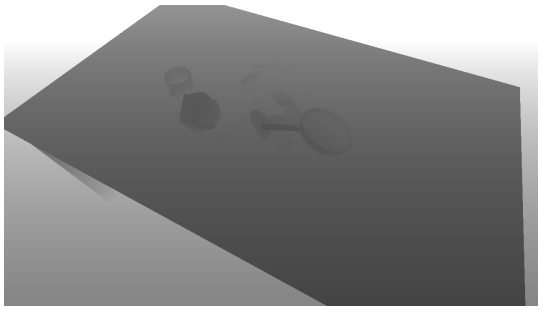
(a) Immagine RGB



(b) Maschera di segmentazione delle parti degli oggetti



(c) Maschera di segmentazione degli oggetti



(d) Immagine di *depth*

Figura 4.2: Esempio di immagini presenti nel dataset relative ad una singola cattura.

4.1.1 Scenario di simulazione

Il Simulation Scenario è il componente del Perception Package attraverso il quale vengono gestite le impostazioni di generazione e cattura dei dati.
È possibile definire il numero di iterazioni che verranno eseguire nella simulazione e il numero di frame per ogni iterazione, che ne determineranno la lunghezza temporale.
Vengono inoltre integrati e gestiti i Randomizer, componenti grazie a cui vengono definiti gli aspetti che verranno randomizzati durante la simulazione. I parametri generati randomicamente fanno riferimento ad un Random Seed definibile dall'utente. Eseguendo due simulazioni e mantenendo invariato il Random Seed, i valori generati durante le simulazioni saranno uguali.

Figura 4.3: Interfaccia base del Simulation Scenario presente nello Unity Editor.

4.1.2 Schema di output SOLO

Un dataset in formato SOLO è composto da immagini e file .json, i quali contengono informazioni aggiuntive non codificabili dalle sole immagini.
Il dataset è organizzato secondo una gerarchia di cartelle, nella quale la cartella al livello più alto è denominata "solo". All'interno della cartella "solo" si troveranno un certo numero di cartelle denominate "sequence.x", tante quanto il numero di iterazioni specificate, e una serie di file .json che descrivono i settaggi impostati per la simulazione. Nello specifico, i file .json contenuti nella cartella "solo" sono:

- annotation_definition.json: contiene la definizione di tutte le annotazioni implementate nel dataset. Nel caso di tale dataset le annotazioni sono i bounding box 2D, la segmentazione delle parti degli oggetti, la segmentazione degli oggetti e la *depth*;
- metadata.json: contiene i dati della simulazione a livello di sistema, ovvero le versioni utilizzate di Unity e del Perception Package, la tipologia di pipeline di render, data e ora in cui la simulazione è stata fatta partire e in cui è terminata, il Random Seed, i Randomizer attivi, i frame catturati e le sequenze eseguite in totale e le annotazioni ricavate dalla simulazione;
- metric_definition.json: contiene l'elenco delle metriche utilizzate, le quali estrapolano dati aggiuntivi riguardanti la simulazione, come l'orientamento delle luci presenti nella scena;
- sensor_definition.json: contiene la definizione dei sensori presenti nella simulazione, rappresentati in Unity dalle camere virtuali.

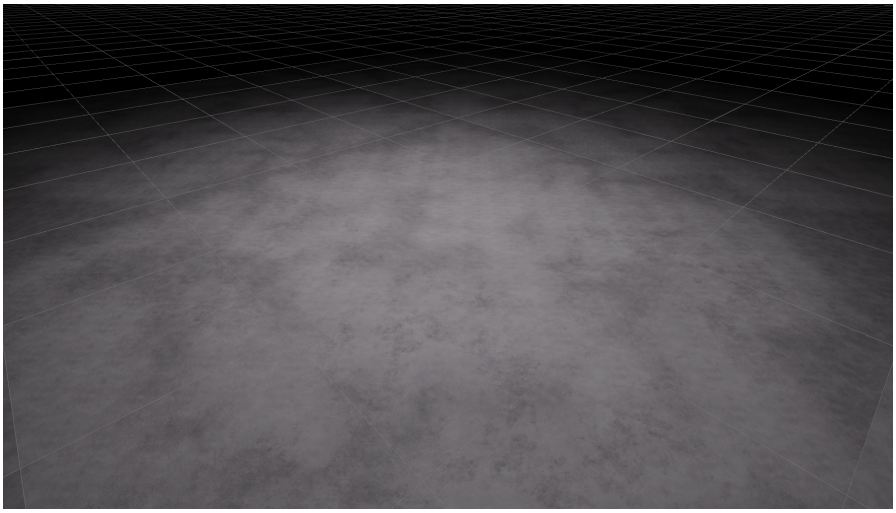
Per quanto riguarda le cartelle "sequence.x", esse contengono i dati raccolti ad ogni iterazione. Una cattura effettuata durante un'iterazione della simulazione porta alla generazione di 4 immagini (definite in 4.1) ed un file .json.
Le informazioni contenute all'interno del file .json sono le seguenti:

- informazioni relative alla camera: posizione, rotazione, risoluzione dell'immagine generata e coordinate della matrice di proiezione prospettica;
- valori dei pixel per i canali RGBA corrispondenti alle maschere di segmentazione degli oggetti presenti nella cattura in coppia al nome della label associata all'oggetto;
- valori dei pixel per i canali RGBA corrispondenti alle maschere di segmentazione delle parti degli oggetti presenti nella cattura in coppia al nome della label associata alla parte dell'oggetto;
- dati relativi ai bounding box 2D relativi agli oggetti presenti nella cattura, sotto forma di coordinate dell'origine e dimensioni del bounding box, in coppia al nome della label associata all'oggetto ed all'id numerico;
- metriche aggiuntive relative alla rotazione della luce, se presente nella scena una luce direzionale.

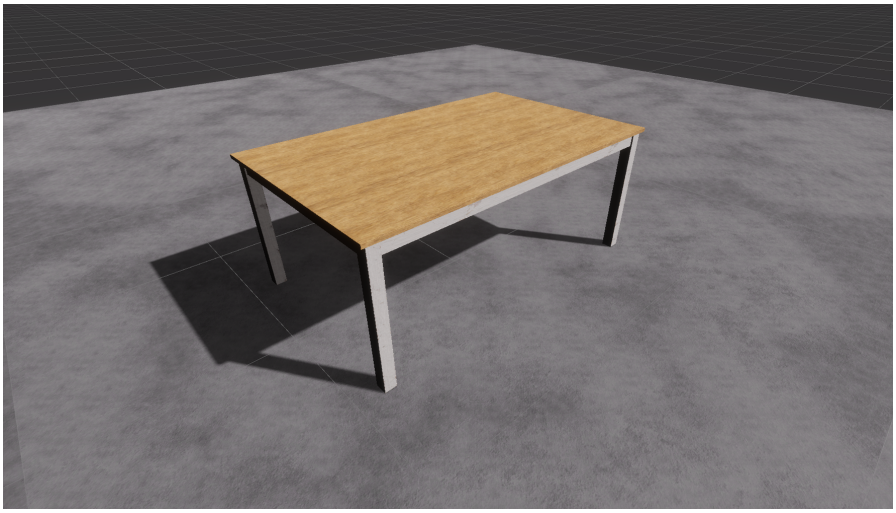
4.2 Differenziazione degli scenari

Il dataset generato è suddiviso in 6 differenti scenari, organizzati con una difficoltà crescente. L'aspetto della difficoltà è rappresentato dal numero di oggetti presenti nella scena, dal loro livello di *clutter*, dalla superficie di appoggio degli oggetti e dalla tipologia di illuminazione della scena.
Gli ambienti realistici creati per ospitare le simulazioni e fungere da superfice di appoggio per gli oggetti istanziati sono due: il primo è costituito da un pavimento di cemento dotato di colorazione uniforme, mentre il secondo è costituito dallo stesso pavimento del primo scenario, sul quale è posizionato un classico tavolo di legno da cucina.
Gli elementi ambientali sono stati modellati su Blender e texturizzati su Substance Painter, per poi essere importati in Unity.

Come si può vedere in figura 4.4, i due ambienti sono illuminati in maniera differente: l'ambiente che presenta solo il pavimento è illuminato da una luce diffusa ed uniforme, mentre l'ambiente con pavimento e tavolo è illuminato da una luce direzionale che genera ombre nette. Di conseguenza, nel primo ambiente gli oggetti saranno illuminati uniformemente ed in modo chiaro, mentre nel secondo gli oggetti saranno soggetti ad ombre che ne renderanno più difficile il riconoscimento. Inoltre,



(a) Pavimento in cemento



(b) Pavimento in cemento e tavolo

Figura 4.4: Ambienti virtuali visti dalla Scene View di Unity.

la luce direzionale del secondo ambiente subirà una variazione di orientamento ogni qual volta un nuovo set di oggetti verrà generato all'interno dell'ambiente, cioè ad ogni nuova iterazione.
Gli scenari sono organizzati in tre macrocategorie, secondo il numero di oggetti istanziati nella scena: Low Objects, Medium Objects e Many Objects. Ognuna di queste macrocategorie sarà implementata per i due ambienti definiti in precedenza, portando così a sei il numero di scenari.
La tabella 4.1 sintetizza le caratteristiche di ciascun scenario.

Il *clutter* indica il modo in cui gli oggetti sono disposti sulla superficie, ne sono state implementate 3 tipologie:

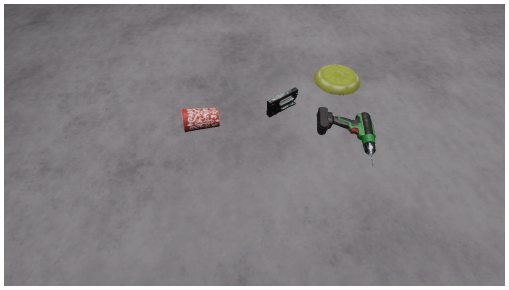
- Any: gli oggetti possono essere disposti in qualunque modo, se il numero di oggetti presenti nella scena è alto, ci saranno oggetti in contatto e sovrapposti tra loro;
- Separated: gli oggetti non devono essere in contatto tra loro, ma devono essere disposti sulla superficie in modo tale da avere dello spazio tra essi;
- Separated + Touching: gli oggetti non devono essere sovrapposti tra loro, bensì devono essere separati o essere in contatto lateralmente, mantenendo la stessa altezza sul piano.

Scenario	Environment	Clutter	Lighting	No. of objects
#00 Low Objects Easy	Uniform Floor	Any	Uniform Light	1-4
#01 Low Objects Hard	Uniform + Table	Any	Random Light	1-4
#02 Medium Objects Easy	Uniform Floor	Separated	Uniform Light	5-7
#03 Medium Objects Hard	Uniform + Table	Any	Random Light	5-7
#04 Many Objects Easy	Uniform Floor	Separated + Touching	Uniform Light	8-12
#05 Many Objects Hard	Uniform + Table	Any	Random Light	8-12

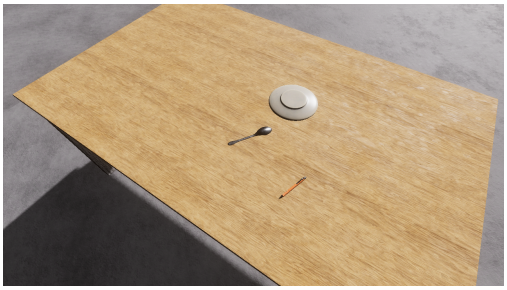
Tabella 4.1: Scenari e loro caratteristiche.

4.3 Scala del dataset

Per determinare la scala del dataset, è stato scelto come punto di riferimento il dataset GraspNet-1Billion, presentato in 2.2.1. Il dataset in questione è costituito da 97.280 immagini e contiene 88 oggetti differenti.
Il dataset generato per questa tesi contiene un totale di 174.000 immagini e 114 oggetti, appartenenti a 38 categorie differenti. Il dataset è suddiviso in una parte di train/validation e in una parte di test.

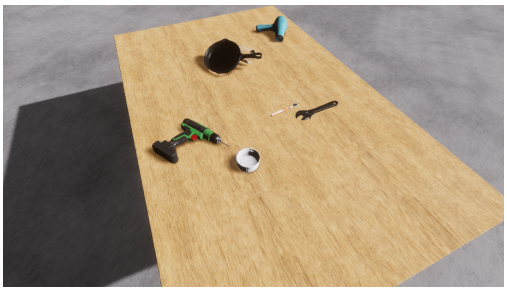


(a) Scenario #00

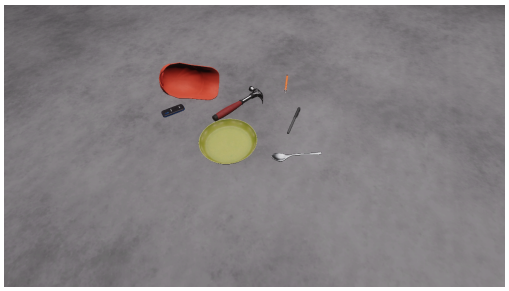


(b) Scenario #01

(c) Scenario #02



(d) Scenario #03



(e) Scenario #04

(f) Scenario #05

Figura 4.5: Immagini RGB provenienti dai 6 scenari.

4.3.1 Train/Validation

Questa porzione di dataset è la più corposa, in quanto composta da 120.000 immagini. Considerando sei scenari e cinque immagini per ogni iterazione, il totale di iterazioni effettuate per ogni scenario è stato pari a 4000. La porzione di train/validation ha utilizzato 63 oggetti, appartenenti a 30 categorie differenti.

4.3.2 Test

La porzione di test contiene in totale 54.000 immagini ed è suddivisa in 3 parti di uguali dimensioni, in base al set di oggetti a cui fanno riferimento:

- oggetti già visti: sono gli stessi oggetti visti in train/validation;

- oggetti simili: sono oggetti diversi rispetto alla prima parte ma appartenenti alle stesse classi e sono in totale 27;
- oggetti sconosciuti: sono oggetti diversi rispetto alle prime due parti sia singolarmente sia come classi di appartenenza. Sono in totale 24 e appartengono a 8 classi differenti.

Ciascuna delle 3 parti contiene 18.000 immagini e per ogni scenario sono state effettuate un totale di 600 iterazioni.

Capitolo 5

Oggetti presenti nel dataset

5.1 Categorizzazione degli oggetti

Gli oggetti presenti all'interno del dataset sono oggetti casalinghi e di uso comune, con dimensioni che vanno da un minimo di 4 cm ad un massimo di 32 cm.
Le categorie a cui appartengono gli oggetti sono divise in due set: il primo, composto da 30 categorie, è stato utilizzato per la porzione di train/validation e per le prime due parti della porzione di test, mentre il secondo, composto da 8 categorie, è stato utilizzato per la terza parte della porzione di test. Le 38 categorie totali comprendono ciascuna dai 2 ai 5 oggetti, le categorie che comprendono più oggetti sono di uso più comune e frequente nell'ambiente casalingo, come si può evincere dalla tabella 5.1.

Categories	No. of objects
Adjustable wrench	3
Book	3
Bottle	3
Can	3
Cap	2
Coffee Mug	4
Comb	3
Common box	2
Credit card	2
Cup	3
Detergent	3
Doughnut	3
Glass	4
Glasses	2

Categories	No. of objects
Hair brush	2
Hairdryer	2
Hammer	3
Knife	5
Magnifying glass	2
Mug	4
Padlock	3
Pan	4
Pen	3
Plate	5
Power drill	3
Scissor	3
Screwdriver	3
Skillet	2
Spoon	5
Stapler	3
Table Fan	3
Teapot	3
Telephone	2
Toothbrush	3
Toy	2
Vegetable	3
Watch	3
Wine glass	3

Tabella 5.1: Categorie di oggetti implementante nel dataset.

5.2 Raccolta degli oggetti

Una volta stilata la lista delle categorie necessarie per la costruzione del dataset, si è passato alla ricerca dei modelli 3D degli oggetti. Il sito web da cui sono stati scaricati la maggior parte dei modelli è Sketchfab[1], sul quale è possibile caricare o scaricare modelli 3D a pagamento o gratuiti, sotto licenza CC BY 4.0[2].
Sketchfab dispone di un potente visualizzatore web per i modelli 3D, da cui è

[1]https://sketchfab.com/feed

[2]https://creativecommons.org/licenses/by/4.0/deed.it

(a) CoffeeMug2 **(b)** Glasses **(c)** RedBellPepper

(d) Screwdriver **(e)** TallGlass2 **(f)** WoodenHammer

Figura 5.1: Esempi di modelli 3D contenuti nel dataset e renderizzati in Blender.

possibile visualizzare il modello texturizzato, la *mesh* del modello e le singole texture map. Questa funzione è risultata essere molto utile, in quanto ha permesso di capire direttamente dal browser se un certo modello potesse essere di buona qualità e adatto al dataset, senza doverlo scaricare ed importare in Blender.
Oltre a Sketchfab, per scaricare i modelli 3D, sono stati utilizzati CGTrader[3] e GazeboSim[4]. CGTrader, in maniera affine a Sketchfab, mette a disposizione modelli 3D caricati da altri utenti, mentre tramite GazeboSim è possibile accedere, tra le altre, alla collezione open source Scanned Objects [16], creata da Google Research appositamente per essere utilizzata in simulazioni 3D nell'ambito della *computer vision*. I modelli 3D contenuti in questa collezione, in totale 1032, sono stati ottenuti tramite la scannerizzazione laser 3D, utilizzando un sistema hardware e software proprietario.
I modelli 3D così ottenuti risultano essere ad alta qualità e fedeli alla realtà.
Per alcune categorie è stato difficile trovare diversi modelli 3D che soddisfacessero i

[3]https://www.cgtrader.com/

[4]https://app.gazebosim.org/GoogleResearch

requisiti di qualità ricercati, si è quindi proceduto alla modellazione degli oggetti mancanti in Blender, tramite la tecnica di *hard surface modeling*.

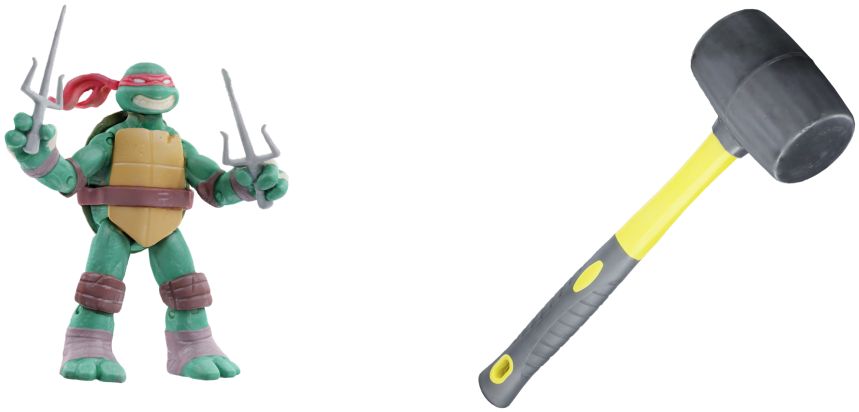


Figura 5.2: Esempi di modelli 3D provenienti da Scanned Objects e renderizzati in Blender.

5.3 Segmentazione per parti

La Perception Camera è un componente incluso nel Perception Package che, una volta aggiunto alla camera virtuale presente nella scena di Unity, permette il riconoscimento degli oggetti e delle label ad essi associati, per poter effettuare la cattura delle immagini e delle maschere. Per consentire alla Perception Camera di rilevare un singolo oggetto come costituito da più parti è essenziale che le singole parti di un oggetto siano identificate nella scena di Unity come Game Object[5] separati.
È stato necessario capire come esportare un determinato oggetto da Blender a Unity, mantenendo la sua identità ma, allo stesso tempo, la suddivisione in più Game Object.
Effettuando diversi tentativi è risultato che il metodo migliore di esportazione fosse quello in formato OBJ, facendo corrispondere le parti dell'oggetto a degli specifici Vertex Group[6] e selezionando come opzione nella finestra di esportazione il raggruppamento per Vertex Group. Questo procedimento porta, una volta

[5]Il Game Object è la classe base a cui ogni entità fa riferimento all'interno di una scena di Unity.

[6]I Vertex Group vengono utilizzati in Blender per etichettare e raggruppare i vertici di una *mesh*.

importato il file OBJ all'interno di Unity, ad un singolo Game Object corrispondente all'oggetto nella sua integrità, il quale racchiude in un rapporto gerarchico tanti Game Object quante sono le parti costituenti dell'oggetto.
I Vertex Group creati in Blender dovranno essere nominati in accordo alla parte a cui fanno riferimento, in modo da ritrovare lo stesso nome all'interno di Unity. Per gestire in modo coerente ed uniforme gli asset all'interno di Unity, gli oggetti che non presentano una suddivisione in parti dovranno comunque avere un Vertex Group che racchiude tutti i vertici della mesh.
Il primo procedimento da svolgere all'interno di Blender è quindi la creazione di un Vertex Group contenente tutti i vertici della *mesh* per ogni oggetto, successivamente saranno creati i Vertex Group specifici per le parti costituenti degli oggetti.
Se per il secondo step è necessario agire manualmente per creare i Vertex Group relativi alle singole parti, per il primo step è possibile automatizzare il processo. A tal proposito, è stato scritto del codice Python che, una volta eseguito nello script editor interno a Blender, genera un Vertex Group contenente tutti i vertici della *mesh* per ogni oggetto selezionato nell'Outliner di Blender, nominandolo allo stesso modo dell'oggetto corrispondente.
Per evitare conflitti all'interno di Unity, in particolar modo nella gestione della fisica, i vertici appartenenti ad uno specifico Vertex Group dovranno essere esclusivi: due Vertex Group diversi non dovranno quindi avere vertici in comune.

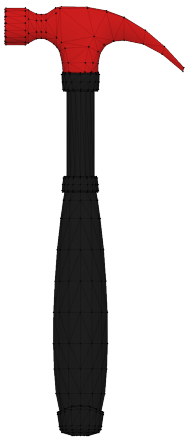


Figura 5.3: Esempio di Vertex Group corrispondente alla testa di un martello.

5.4 Texturing

La maggior parte dei modelli 3D reperiti da Sketchfab e CGTrader comprendono le texture map necessarie ad ottenere un buon risultato nell'ottica del fotorealismo. In alcuni casi è stato però necessario agire manualmente per realizzare nuovamente il materiale dell'oggetto, in quanto risultava mancante o di bassa qualità. Per questi casi, così come per gli oggetti modellati interamente su Blender, è stato utilizzato Substance 3D Painter, analizzato in 3.3.
Il primo passaggio è stato verificare che l'*UV unwrapping*[7] fosse ottimizzato correttamente, in modo da ottenere il miglior risultato possibile all'interno di Substance 3D Painter. Successivamente è stato creato un materiale fotorealistico che potesse adattarsi al meglio al modello, sfruttando sia la vasta libreria di materiali forniti nel software, sia creando nuovi materiali combinando layer di base.
I modelli 3D provenienti dal dataset Scanned Object dispongono solamente della texture di *base color*, non sufficiente per ottenere un risultato fotorealistico. Per risolvere questo problema è stato sfruttato il software Substance 3D Sampler, analizzato in 3.4, il quale è in grado di generare le texture di *roughness* e *normal* ricevendo in input la texture di *base color*. Il software non è in grado di generare automaticamente la texture *metallic*, ma offre l'opzione di applicare a mano una maschera sull'oggetto, per indicare quali parti debbano essere di tipo metallico e quali no.
Una funzione molto interessante offerta da Substance 3D Sampler è quella di Delighting, che permette di ridurre i riflessi luminosi provenienti da fonti di luce esterne. Si è rivelato essere molto utile per i modelli 3D prelevati da Scanned Objects in quanto, soprattutto sugli oggetti con materiali più lucidi, erano presenti i riflessi delle luci utilizzate per illuminare gli oggetti durante il processo di scansione 3D. Questi riflessi sono rimasti sulla texture di *base color*, andando a falsare la risposta della luce del materiale all'interno di un ambiente virtuale, che presenta riflessi provenienti da luci non effettivamente localizzate nella scena.

5.4.1 Gestione delle texture in Unity

La Unity HDRP, analizzata in 3.1, non utilizza un formato standard per applicare le texture ai modelli 3D, bensì utilizza un formato basato sulla compressione di più immagini in scala di grigi all'interno dei 4 canali di una singola immagine RGBA. In particolare, si fa riferimento alla *Mask map*, la quale contiene nel canale R la texture *metallic*, nel canale G la texture *ambient occlusion*, nel canale B la texture *detail mask* e nel canale A la texture *smoothness*.

[7]Procedimento per cui la *mesh* di un modello 3D viene scomposta e "distesa" su un piano, in modo tale da poter applicare senza distorsioni la texture 2D.

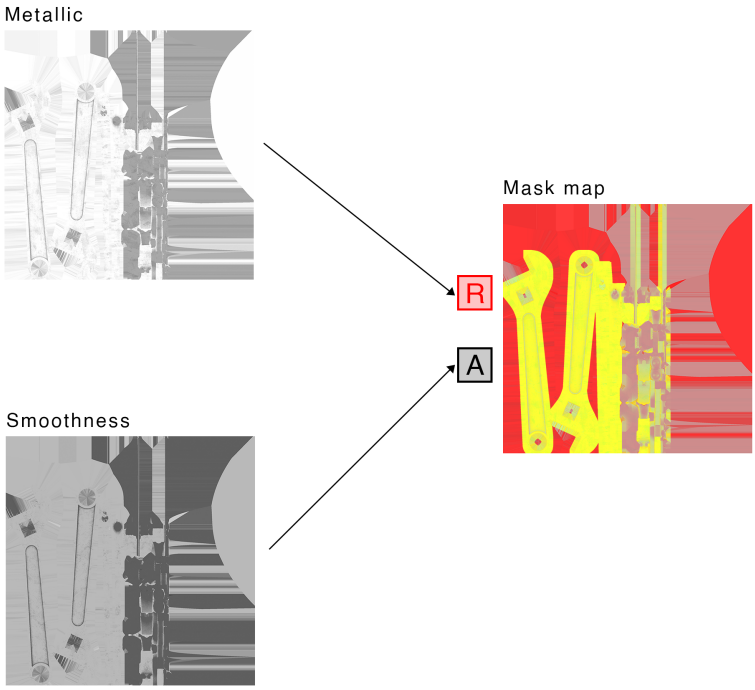


Figura 5.4: Formattazione della Mask map in Unity.

La texture *smoothness* è una texture di *roughness* con i valori invertiti, ovvero ad un pixel nero corrisponde un pixel bianco e viceversa, comprendendo tutti i valori di grigio intermedi. Questo tipo di formattazione è stato schematizzato in figura 5.4. Nel caso del lavoro svolto per questa tesi, non è stato necessario l'utilizzo delle texture *ambient occlusion* e *detail mask*.
Per ottenere delle *mask map* conformi tra loro ed evitare problemi di incompatibilità con Unity, sono stati utilizzati Substance 3D Painter e Substance 3D Sampler per esportare le texture nel formato corretto e adatto per essere importate nel miglior modo all'interno di Unity.

5.5 Preparazione ed importazione degli asset in Unity

Per gestire tutti i modelli 3D degli oggetti e renderli conformi tra loro prima dell'importazione in Unity è stato creato un progetto di Blender per collezionare gli oggetti durante la ricerca.
Per ogni oggetto è stata regolata la rotazione e la scala, rispettivamente con valori di 0° e 1.0 per i tre assi X, Y e Z.

Gli oggetti sono stati ridimensionati, se necessario, in modo che le dimensioni all'interno di Blender fossero comparabili a quelle dell'oggetto nel mondo reale. Le dimensioni fisiche impostate in Blender sono mantenute nel passaggio a Unity.
Passando alla modalità Edit Mode di Blender, è stato verificato che le *mesh* dei modelli 3D fossero regolari, che le normali dei poligoni puntassero verso l'esterno e che non ci fossero vertici sovrapposti tra loro. Per verificare l'ultimo punto si è usufruito del comando "Merge by distance", il quale effettua un *merge* dei vertici le cui coordinate spaziali sono identiche o la cui differenza è minore di una soglia molto bassa, in quanto vertici sovrapposti potrebbero portare a problematiche durante le simulazioni fisiche all'interno di Unity. È stato inoltre verificato che la *mesh* dei modelli 3D non fosse eccessivamente dettagliata. Un numero di vertici troppo elevato è, infatti, causa di rallentamenti durante una simulazione e, poiché il numero di simulazioni da effettuare per generare il dataset è nell'ordine delle migliaia, si è cercato di ottimizzare al massimo le tempistiche delle singole simulazioni.
Un ulteriore accorgimento preso è stato quello di uniformare la posizione dell'origine degli oggetti in Blender, centrando la posizione dell'origine rispetto al centro di massa. L'origine di un oggetto è il punto utilizzato per calcolare ed applicare le trasformazioni: è stato quindi scelto un punto che fosse riconducibile alla struttura fisica dell'oggetto.
Il formato scelto per l'esportazione è il formato basato su ASCII OBJ, creato da Wavefront Technologies tra gli anni '80 e '90 per visualizzare i modelli 3D all'interno del loro software Advanced Visualizer [17].
Un modello 3D in formato OBJ viene esportato in coppia ad un file in formato MTL (Material Template Library), che contiene informazioni riguardanti il colore e le texture ad esso associate. Importando insieme il file OBJ ed il file MTL, Unity riconoscerà il materiale e le texture incorporate al modello 3D. Una volta estratto il materiale, esso potrà essere applicato al modello 3D. La funzione "Extract material" consente di aggiungere ai propri asset un file con estensione .mat relativo al materiale del modello 3D, rendendolo così indicizzabile. Unity non supporta l'estrazione e la modifica di materiali selezionando più modelli 3D insieme, per cui è stato implementato lo script BatchExtractMaterials.cs[8], fornito dall'utente *yasirkula* sulla piattaforma GitHub. L'implementazione di questo script consente, tramite un'interfaccia grafica che può essere aggiunta allo Unity Editor, l'estrazione dei materiali associati a tutti i modelli 3D che vi vengono trascinati, all'interno di una specifica cartella. È inoltre possibile rimappare i materiali, nel caso in cui esistano già dei materiali nominati come quelli che devono essere estratti, in modo da sostituirli ed applicare ai modelli 3D i nuovi materiali.
Il materiale creato di default da Unity è di tipo opaco, adattabile alla maggior

[8]https://gist.github.com/yasirkula/3d3ffe9fdf6330a1328dd026b9f1cc62

parte degli oggetti appartenenti al dataset. Per gli oggetti dotati di materiale in vetro o trasparente si è agito per modificare il materiale, andando a cambiare il valore del campo Surface Type da "Opaque" a "Transparent".
Sono stati inoltre modificati i valori di Refraction Model, impostando il valore "Sphere", e di Index of Refraction per ottenere un risultato il più possibile realistico, nei limiti dell'elevato calcolo computazionale comportato dal rendering di materiali trasparenti.
Per ogni tipo di immagine che viene importata all'interno di Unity è possibile modificare una serie di impostazioni, per adattarle al meglio alla loro funzione. Nel caso delle texture di *normal* è necessario modificare il campo Texture Type da "Default" a "Normal map", in modo tale che Unity interpreti correttamente le informazioni contenute nella texture. Questa operazione non è da svolgere per gli altri tipi di texture, che dovranno rimanere nello stato di default.

5.5.1 Creazione dei Prefab

I modelli 3D degli oggetti che devono essere generati durante la costruzione del dataset sono dotati, all'interno di Unity, di una serie di componenti che offrono funzionalità aggiuntive, le quali espandono le funzionalità base dei Game Object. Nel momento in cui un Game Object deve essere sempre istanziato con una serie di componenti predefinite, il workflow standard di Unity è quello di generare un Prefab relativo al Game Object e ai componenti ad esso associati.
I Prefab consentono di gestire asset definiti e riutilizzabili, posizionati all'interno di una specifica cartella del progetto Unity, comportandosi come un template di un determinato Game Object. Modificando un Prefab, vengono di conseguenza modificate tutte le sue istanze presenti nella scena di Unity, evitando così di dover applicare manualmente la modifica per ogni istanza.
È possibile creare Prefab annidati, funzionalità utile per questo lavoro in quanto, come specificato in 5.3, le parti separate di uno stesso oggetto saranno organizzate in un rapporto gerarchico di Game Object distinti.

Capitolo 6

Generazione del dataset

Il dataset generato per questa tesi basa le sue fondamenta sulla cattura di immagini all'interno di un ambiente virtuale contenente un certo numero di oggetti sparsi in modo casuale su una superficie piana. La casualità nella disposizione degli oggetti è ottenuta tramite una simulazione fisica, facendo cadere gli oggetti sul piano da una determinata altezza. Il ciclo elementare di simulazione è strutturato in tre passaggi di base:

- generazione degli oggetti;
- caduta degli oggetti ed assestamento della simulazione fisica;
- cattura delle immagini.

Una volta che le immagini sono state catturate gli oggetti istanziati vengono rimossi dalla scena per dare spazio agli oggetti successivi. Questo ciclo si ripete per il numero di volte specificato nel campo "Iteration count" del componente Simulation Scenario del Perception Package.

6.1 Gestione del procedimento di generazione del dataset

Il Perception Package offre tutti i componenti necessari per attuare la generazione degli oggetti, la cattura delle immagini e la cattura dei dati. Questi componenti sono basati su script C#, dunque completamente personalizzabili secondo le proprie esigenze. La generazione degli oggetti è affidata al Simulation Scenario, tramite cui è possibile gestire i parametri principali della simulazione, mentre la cattura delle immagini è attuata mediante la Perception Camera. È possibile inserire all'interno della scena Unity più camere virtuali impostate come Perception Camera, ma è

consentito che un solo Simulation Scenario sia attivo nel momento in cui si avvia una simulazione.
Per una gestione ottimale dei differenti scenari ed un'organizzazione chiara del progetto di Unity, è stata creata una scena di Unity dedicata per ogni scenario. La suddivisione in scene diverse permette di avere un quadro chiaro dei parametri impostati per ogni scenario, oltre ad alleggerire il carico computazionale al momento in cui viene avviata una simulazione.

6.1.1 Simulation Scenario

Il Simulation Scenario ha la responsabilità di gestire l'andamento della simulazione, attraverso i suoi parametri, e di gestire la lista dei Randomizer ad esso associati, unità di controllo che introducono variabilità nel dataset in merito ad aspetti specifici della simulazione. Per rendere attivo un Simulation Scenario nell'ambito di una scena Unity è necessario aggiungere ad un oggetto Empty il componente Fixed Length Scenario. I parametri che questo componente permette di modificare direttamente dall'Editor di Unity sono i seguenti:

- "Frames per Iteration", il numero corrispondente alla lunghezza temporale espressa in frame di ogni iterazione della simulazione;
- "Random Seed", un numero su cui si basa la generazione randomica di valori legati alla simulazione;
- "Start Iteration", l'indice della prima iterazione che verrà eseguita;
- "Iteration Count", il numero di iterazioni che verranno eseguite nell'arco della simulazione.

Viene offerta la possibilità di esportare una determinata configurazione di scenario in formato JSON, in modo da poter importare diverse configurazioni complete, eventualmente anche a *runtime*. Una volta avviata la simulazione, l'interfaccia del Simulation Scenario mostra una barra di caricamento che indica la percentuale di completamento del processo, dettaglio utile nel momento in cui si esegue una simulazione voluminosa.

6.1.2 Perception Camera

Il componente Perception Camera deve essere aggiunto ad una camera virtuale presente all'interno della scena di Unity. La sua funzione è quella di catturare le immagini RGB e tutti i dati di *ground-truth* relativi alla vista della camera virtuale, che andranno a comporre il dataset. È possibile associare un ID ed una descrizione, nell'eventualità in cui più camere siano presenti nella stessa scena.

Il campo Capture Trigger Mode si riferisce alla modalità di acquisizione delle immagini e può assumere due valori: "Manual" o "Scheduled". Nella configurazione "Manual" la Perception Camera effettuerà la cattura di un'immagine nel momento in cui via script sarà invocato il metodo RequestCapture, mentre nella configurazione "Scheduled" la cattura verrà effettuata ad intervalli di frame regolari. Durante i test preliminari di generazione del dataset si è notato che nella configurazione "Scheduled", col passare delle iterazioni, veniva accumulato un errore temporale che rendeva incostanti gli intervalli di cattura delle immagini. È stato quindi deciso di adottare la configurazione "Manual" per avere il controllo totale degli istanti in cui effettuare le catture.

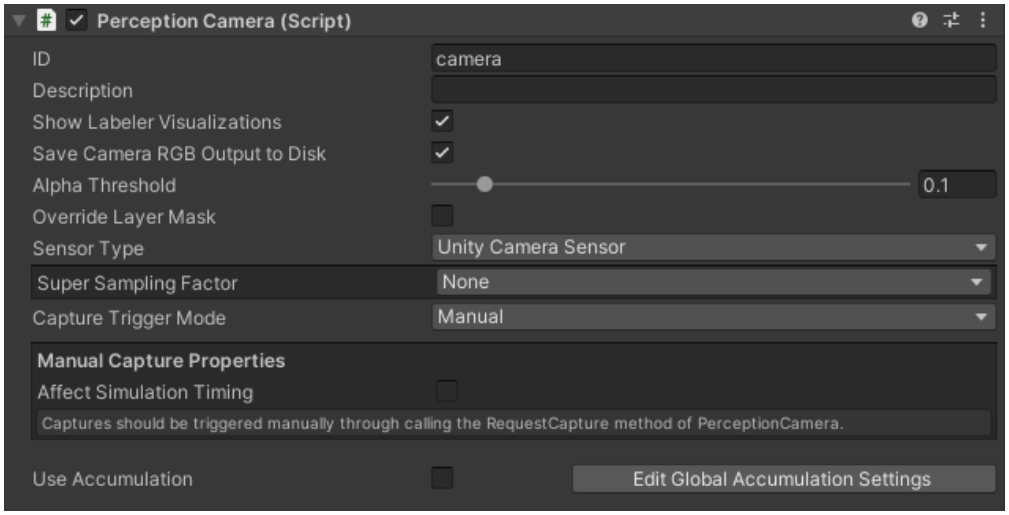


Figura 6.1: Interfaccia del componente Perception Camera vista dallo Unity Editor.

6.1.3 Labeler

Il componente Perception Camera implementa la gestione dei Labeler, elementi fondamentali per la generazione del dataset in quanto consentono di estrapolare le informazioni di *ground-truth* per ogni frame catturato dalla camera. La lista dei Labeler associati alla Perception Camera determinerà la tipologia di informazioni contenute nell'output del dataset, potendo scegliere tra i Labeler predefiniti forniti dal Perception Camera oppure creandone di personalizzati tramite uno script C#. In figura 6.2 vengono mostrati i Labeler utilizzati nella creazione del dataset.
I Labeler recepiscono le informazioni di *ground-truth* dalle label associate agli asset 3D presenti nella scena. Le label vengono aggiunte manualmente e contengono informazioni semantiche riguardo all'oggetto specifico. Per associare una label

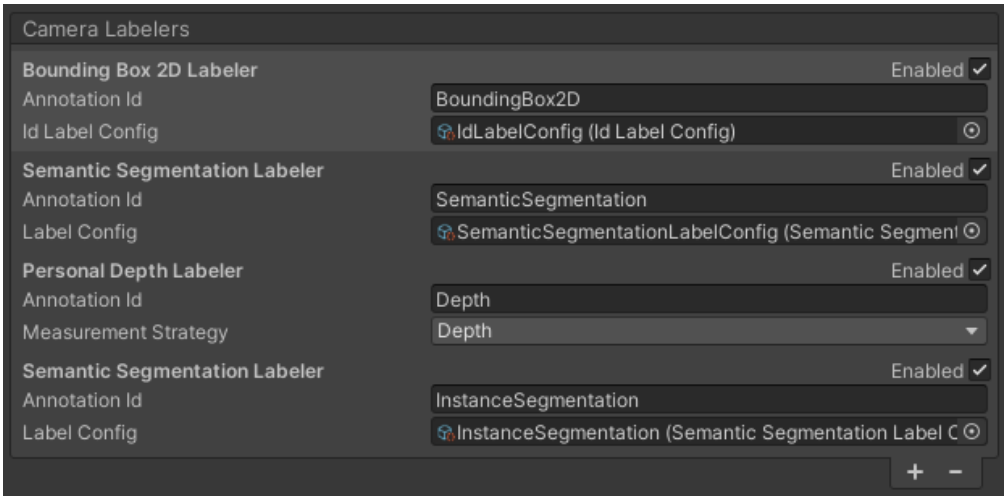


Figura 6.2: Lista dei Labeler associati alla Perception Camera.

ad un oggetto è sufficiente aggiungere al GameObject più interno, ovvero quello corrispondente alla segmentazione delle parti, il componente Labeling. Dall'interfaccia di questo componente è possibile creare una label usando il nome dell'asset e questa operazione può essere svolta selezionando più asset contemporaneamente, velocizzando il processo di labeling.
Tutte le label associate agli oggetti sono raccolte in specifici file, denominati Label Config, da cui i Labeler attingono le informazioni. Sarà dunque presente un file di Label Config per ogni Labeler associato alla Perception Camera. I file di Label Config sono strutturati come un elenco dove ogni elemento presenta l'associazione tra il nome della label e un colore, nel caso della maschere di segmentazione (figura 6.3), o un ID numerico, nel caso dei bounding box 2D. In figura 6.2 si possono notare i file di Label Config associati ad ogni Labeler.

Immagini di depth

La generazione delle immagini di *depth* è inclusa di default all'interno del Perception Package, aggiungendo alla lista di Labeler della Perception Camera il Depth Labeler. Mantenendo questa configurazione standard viene generato un file di output con estensione .exr, visualizzabile tramite un software dedicato, come ad esempio RenderDoc[1]. Il formato dell'immagine di *depth* così generata è a 32 bit e viene utilizzato esclusivamente il canale R per immagazzinare i valori di *depth* associati ai

[1]https://renderdoc.org/

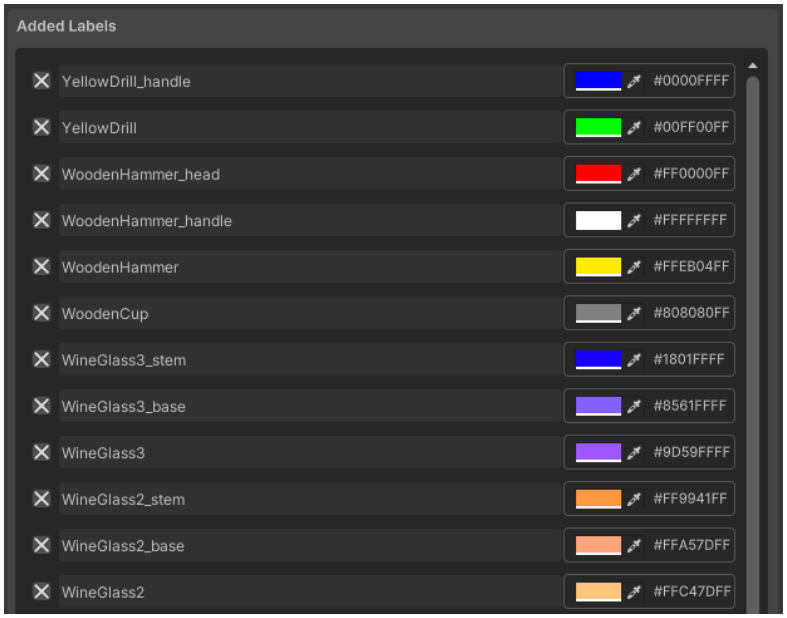


Figura 6.3: Porzione del file Label Config relativo alla segmentazione delle parti degli oggetti.

pixel. L'immagine visualizzata su RenderDoc risulta quindi essere monocromatica con differenti valori di rosso, come si può vedere in figura 6.4.

Per uniformare gli output del dataset e rendere più accessibile la visualizzazione delle immagini di *depth* è stato necessario intervenire manualmente per ottenere immagini con estensione .png e in scala di grigi.
Per quanto riguarda l'estensione .png è stato modificato lo script C# relativo al Depth Labeler, nella sezione in cui veniva specificato l'encoding dell'immagine di output.
Per ottenere un'immagine in scala di grigi è stato invece modificato lo *shader* utilizzato per associare i valori di distanza tra la camera virtuale e la scena ai valori dei pixel dell'immagine di output. Il valore di ritorno dello *shader* consisteva originariamente in un valore espresso in `float4`, dove i 4 parametri fanno riferimento ai canali R,G,B ed A. Per convertire l'output ad un'immagine in scala di grigi è stato sufficiente modificare il valore di ritorno associando per ogni pixel lo stesso valore ai canali R,G e B.

6.2 Generazione degli oggetti

Ogni iterazione inizia con la generazione di un numero di oggetti variabile a seconda dello scenario al momento attivo. Gli oggetti vengono generati ad una determinata

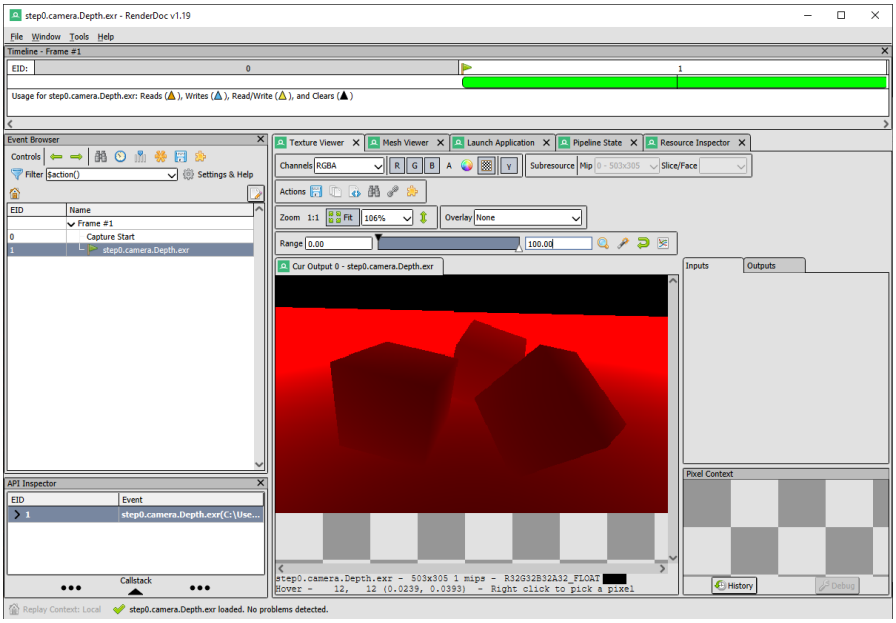


Figura 6.4: Esempio di immagine di *depth* di default in formato .exr visualizzata all'interno di RenderDoc [18].

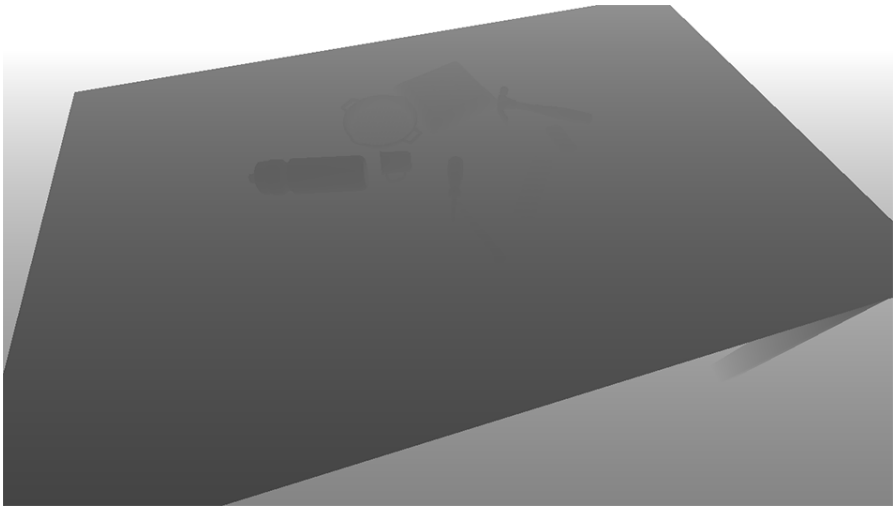


Figura 6.5: Immagine di *depth* contenuta all'interno del dataset.

altezza, ognuno con una rotazione casuale attorno al proprio centro, per poi cadere sul piano di appoggio disponendosi in maniera casuale.
La generazione degli oggetti è da considerarsi come elemento che introduce variabilità all'interno del dataset, dunque è un aspetto che nel Perception Package viene controllato e parametrizzato da un Randomizer dedicato. I Randomizer sono unità

di controllo che vengono aggiunte al Simulation Scenario, secondo l'ordine desiderato di attivazione: un Randomizer verrà infatti influenzato da quello precedente e, allo stesso modo, influenzerà il successivo. Lo scopo di questi componenti è quello di controllare le attività che dovranno essere pilotate da valori randomizzati durante il corso della simulazione. I Randomizer ricevono i valori da componenti denominati Sampler, i quali generano dei valori *float* all'interno di uno specifico range effettuando il campionamento di una distribuzione di probabilità.

6.2.1 Foreground Object Placement Randomizer

Il Randomizer che si occupa della generazione degli oggetti è denominato Foreground Object Placement Randomizer. Essendo editabile tramite script C#, sono state effettuate diverse modifiche per adattarlo al meglio alle esigenze di simulazione. Il funzionamento originario di questo Randomizer prevede l'istanziazione di un numero indeterminato di asset posti frontalmente rispetto alla camera virtuale, all'interno di un'area delimitata e orientata verticalmente nello spazio.
I parametri modificabili sono limitati: è possibile modificare l'estensione dell'area di generazione lungo gli assi X ed Y e la sua profondità lungo l'asse Z.
Per specificare gli asset da istanziare viene indicata la cartella contenente i prefab desiderati.
È infine possibile modificare la distanza di separazione minima tra i centri degli asset generati.

Il primo intervento eseguito è stato ruotare il piano su cui vengono istanziati gli oggetti, rendendolo parallelo al piano di appoggio sottostante. È stato modificato il codice relativo al posizionamento degli oggetti, scambiando tra loro le coordinate Y e Z. In questo modo le dimensioni del piano di posizionamento fanno riferimento agli assi X e Z, mentre il parametro Depth non fa più riferimento ad una profondità ma ad un'altezza, il suo nome è stato quindi cambiato in Height.
Gli scenari che compongono il dataset elaborato per questa tesi, elencati nella tabella 4.1, includono un numero di oggetti variabile, espresso sotto forma di range tra un minimo ed un massimo di oggetti istanziabili per ogni iterazione. Il Foreground Object Placement Randomizer nella sua configurazione di base non prevede di poter fissare un limite al numero di oggetti generati, bensì viene istanziato il maggior numero di oggetti possibile in accordo all'estensione della superficie di posizionamento e alla distanza di separazione. Per implementare la generazione di un numero di oggetti contenuto in un range, sono stati aggiunti al Randomizer due parametri, "Spawn Min" e "Spawn Max". Questi due nuovi parametri accettano valori interi che vengono utilizzati all'interno della funzione `Random.Range(int.minInlcusive, int.maxExclusive)`, la quale restituisce un valore intero randomico, incluso nel range specificato, denominato `spawnLimit`. La

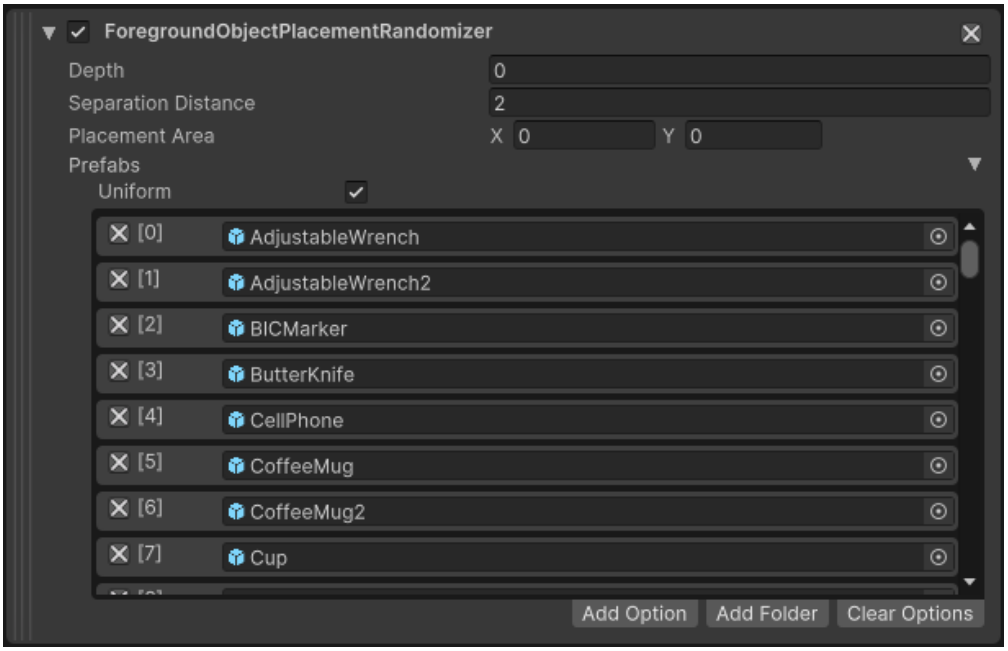


Figura 6.6: Interfaccia di base del Foreground Object Placement Randomizer.

scelta degli oggetti da istanziare avviene all'interno di un ciclo `for` e nel momento in cui un contatore raggiunge il valore `spawnLimit`, il ciclo `for` viene interrotto.

Gestione della ripetizione di oggetti

Il Foreground Object Placement Randomizer dispone di un insieme limitato di prefab da cui è possibile attingere per la fase di generazione. Di conseguenza, è possibile che si verifichino delle occorrenze ripetute di oggetti all'interno della stessa iterazione. Questa condizione è particolarmente restrittiva per gli scenari caratterizzati da un basso numero di oggetti e per le sezioni di test che attingono dal set di oggetti simili (comprendente 27 oggetti) e dal set di oggetti sconosciuti (comprendente 24 oggetti). Questa problematica porterebbe il dataset a non sfruttare in modo ottimale gli oggetti a disposizione, pertanto è stata integrata allo script del Randomizer una funzione di controllo, denominata `CheckDuplicate`. Questa funzione viene chiamata durante l'esecuzione del ciclo `for` associato alla fase di generazione degli oggetti e verifica che l'oggetto appena creato non sia già presente tra gli oggetti precedentemente istanziati. In caso di rilevamento di un duplicato, si procede alla rimozione dello stesso, per poi generarne uno nuovo che rispetti il requisito di univocità relativo all'iterazione corrente.

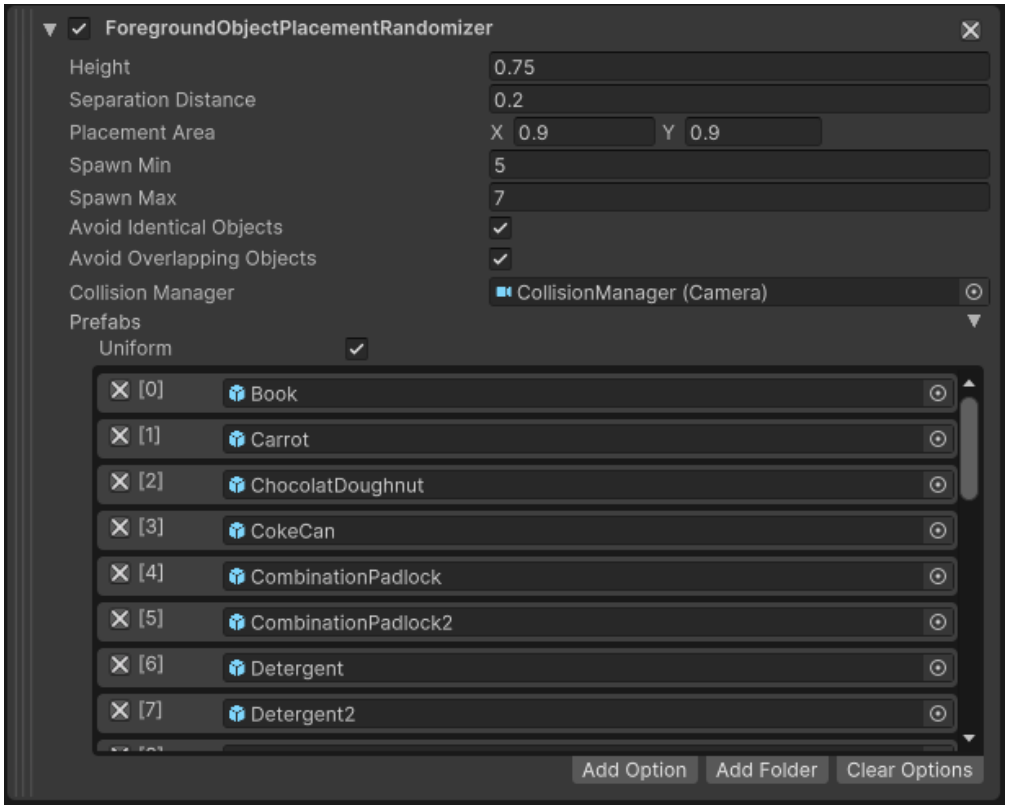


Figura 6.7: Interfaccia personalizzata del Foreground Object Placement Randomizer.

Il blocco della generazione di oggetti identici all'interno della stessa iterazione può essere abilitato o disabilitato attraverso la selezione dell'opzione "Avoid Identical Objects" presente nell'interfaccia personalizzata del Foreground Object Placement Randomizer.

Gestione della sovrapposizione di oggetti

I sei distinti scenari implementati nel dataset sono caratterizzati da un aspetto chiave, ossia la densità di *clutter*, che indica il grado di sovrapposizione e prossimità degli oggetti posizionati all'interno della scena. Facendo riferimento alla tabella 4.1, si distinguono tre differenti impostazioni di *clutter*: Any, utilizzata negli scenari #00, #01, #03 e #05, Separated, utilizzata nello scenario #02 e Separated + Touching, utilizzata nello scenario #04.
Per quanto riguarda l'impostazione Any, non è stato necessario alcun intervento poiché questa corrisponde ad una disposizione completamente casuale degli oggetti sulla superficie di supporto. Tuttavia, è stato necessario sviluppare una soluzione

per garantire l'attuazione dell'impostazione Separated, che è stata in seguito estesa alla casistica Separated + Touching.
Il sistema impiegato per correggere la sovrapposizione tra oggetti si basa sulla fisica, ed in particolare sull'applicazione di forze specifiche agli oggetti rilevati come sovrapposti. Il fulcro di questo sistema è costituito da una camera virtuale alla quale è stato associato uno script C# appositamente creato per questa finalità. Questo componente è stato denominato Collision Manager.
La camera virtuale è posizionata centralmente rispetto alla scena e fornisce una visuale dall'alto verso il basso, con il proprio campo visivo (FOV, field of view) configurato in modo da ricoprire integralmente lo spazio occupato dagli oggetti all'interno della scena.

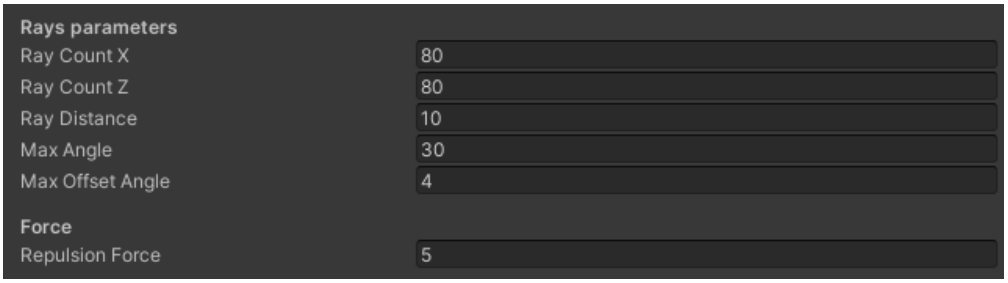


Figura 6.8: Interfaccia del componente Collision Manager all'interno dello Unity Editor.

Per individuare la sovrapposizione di oggetti è stata implementata una funzione interna a Unity denominata `Physics.Raycast`. Questa funzione esegue il *casting* di un raggio e raccoglie informazioni sulle collisioni tra il raggio stesso ed altri elementi presenti nella scena. È stato definito il parametro "Max Angle" del Collision Manager, il quale rappresenta un valore angolare in gradi. Questo angolo, applicato alle direzioni X e Z, definisce il volume all'interno del quale verranno generati i raggi.
Il parametro "Ray Distance" è richiesto dalla funzione `Physics.RayCast` e corrisponde alla massima distanza per cui il raggio può rilevare una collisione. Il numero di raggi da generare è definito dal parametro "Ray Count", espresso in modo indipendente per le direzioni lungo l'asse X e l'asse Z.
Una volta che gli oggetti sono caduti sulla superficie di appoggio e la loro posizione diventa statica, vengono generati i raggi per un periodo di 100 frame. Ad ogni frame ciascun raggio subisce uno sfalsamento casuale espresso in gradi, con il range di questo sfalsamento definito dal parametro "Max Offset Angle" del Collision Manager.
L'introduzione dello sfalsamento casuale ha lo scopo di creare disordine all'interno di ciò che altrimenti sarebbe una griglia uniforme di raggi, la quale potrebbe

non individuare tutti i punti di sovrapposizione tra gli oggetti. L'utilizzo dello sfalsamento mira inoltre ad evitare la generazione di un numero eccessivo di raggi, il che renderebbe più precisa la rilevazione delle sovrapposizioni ma aumenterebbe la complessità della simulazione, compromettendone l'ottimizzazione.
È stata creata una lista di tipo `RayCastHit` con l'obiettivo di registrare le collisioni di ciascun raggio con i Game Object contrassegnati con il tag "GraspObject", associato ai prefab corrispondenti agli oggetti da istanziare nella scena. Durante questa fase è stata incrementata una variabile contatore per ogni collisione tra il raggio e un oggetto. La rilevazione di una sovrapposizione tra oggetti si verifica quando un raggio entra in collisione con più di un oggetto, ovvero quando il valore della variabile contatore associata a quel raggio è maggiore o uguale a 2.
Per evitare la rilevazione di una sovrapposizione quando un raggio colpisce solo una parte di un oggetto e successivamente esce dalla collisione per poi collidere nuovamente con lo stesso oggetto, è stato implementato un controllo per verificare che gli oggetti coinvolti nella collisione col raggio abbiano nomi e posizioni nello spazio differenti.
Una volta accertata l'effettiva sovrapposizione tra due oggetti, si procede con la fase di repulsione reciproca degli oggetti coinvolti. Vengono calcolate due variabili di tipo `Vector3` ottenute dalle differenze tra le posizioni dei due oggetti. Queste variabili corrispondono ai vettori spaziali che collegano i centri dei due oggetti nelle direzioni opposte. Successivamente, vengono ricavati i componenti `Rigidbody` dei due Game Object tramite la funzione `GetComponent`, poiché necessari per la simulazione fisica. Mediante la funzione `AddForce`, viene applicata ai `Rigidbody` una forza con intensità specificata dal parametro "Repulsion Force", in direzione opposta rispetto all'altro oggetto. L'applicazione di queste forze opposte spingerà i due oggetti in direzioni contrarie, eliminando così la sovrapposizione. È stata scelta un durata temporale di 100 frame per consentire al sistema di gestire eventuali ulteriori sovrapposizioni che potrebbero verificarsi a causa dello spostamento degli oggetti sulla superficie d'appoggio.
Per adattare questa tecnica all'impostazione Separated + Touching è stato sufficiente ridurre la forza di repulsione agendo sul parametro "Repulsion Force" del Collision Manager. Questo parametro è stato ridotto da un valore di 5, utilizzato per l'impostazione Separated, ad un valore di 1.
Poiché una sovrapposizione è rilevata da molteplici raggi, le forze applicate agli oggetti saranno tendenzialmente più di una. Nel caso dell'impostazione Separated, pochi input di forza sono sufficienti per allontanare quasi istantaneamente gli oggetti. Tuttavia, per l'impostazione Separated + Touching, si verificheranno un numero maggiore di input di forza che causeranno piccoli spostamenti negli oggetti. Questi piccoli spostamenti porteranno gli oggetti in uno stato in cui i raggi non rileveranno più una sovrapposizione, ma gli oggetti rimarranno comunque molto vicini. Questo approccio consente di eliminare la sovrapposizione mentre si mantiene la condizione

di vicinanza richiesta per l'impostazione Separated + Touching.
L'attivazione del meccanismo per evitare le sovrapposizioni tra oggetti può essere abilitato o disabilitato mediante la spunta dell'opzione "Avoid Overlapping Objects", presente nell'interfaccia personalizzata del Foreground Object Placement Randomizer.

6.2.2 Rotation Randomizer

Il secondo componente Randomizer inserito nella gerarchia all'interno del Simulation Scenario corrisponde al Rotation Randomizer. La sua funzione è quella di applicare una rotazione casuale agli oggetti nel momento in cui vengono istanziati dal Foreground Obejct Placement Randomizer. I parametri configurabili del Rotation Randomizer includono il range di valori di rotazione per ciascun asse, insieme alla tipologia di distribuzione di probabilità da cui verranno estratti i valori casuali. L'importanza di questo componente risiede nel contribuire a creare una disposizione più caotica e realistica degli oggetti durante la simulazione.
Senza l'intervento del Rotation Randomizer gli oggetti cadrebbero sulla superficie di appoggio e rimarrebbero immobili, formando una disposizione relativamente ordinata. L'applicazione di una rotazione casuale prima dell'impatto con la superficie comporta una fase di assestamento dopo il contatto, durante la quale gli oggetti ruotano per trovare una posizione di equilibrio, interagendo tra loro.
Per definire quali oggetti devono essere sottoposti al processo di rotazione casuale da parte del Rotation Randomizer, è necessario aggiungere lo script "Rotation Randomizer Tag" a ciascun prefab degli oggetti. Durante l'esecuzione della simulazione, il Rotation Randomizer influenzerà solamente sugli oggetti presenti nella scena che sono dotati di questo componente, applicando loro la rotazione casuale per i tre assi. Gli oggetti sprovvisti del "Rotation Randomizer Tag" non saranno soggetti a tale manipolazione e manterranno il loro orientamento iniziale.

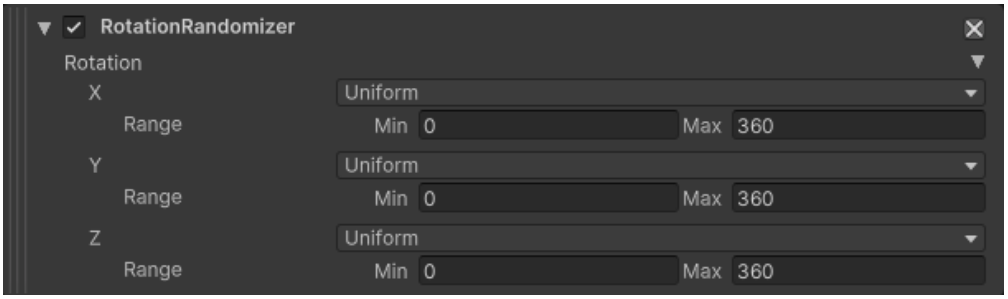


Figura 6.9: Interfaccia del Rotation Randomizer.

6.2.3 Light Randomizer

La metà degli scenari implementati all'interno del dataset, nello specifico gli scenari #01, #03 e #05, presentano come caratteristica distintiva, nella categoria "Lighting", il termine "Random Light". Questi scenari si contraddistinguono per un'illuminazione direzionale e ombre nette, facendo uso della tipologia di luce "Directional Light" offerta da Unity. Durante il corso della simulazione, ad ogni iterazione, la luce subisce una rotazione casuale ed una variazione dell'intensità luminosa, illuminando la scena in modo differente. L'intervento sull'illuminazione delle scene è stato finalizzato ad introdurre varietà e ad aggiungere un elemento di difficoltà all'interno del dataset.
Questa tipologia di Randomizer non fa parte dei Randomizer di base e per implementarlo è stata seguita la guida presente nella documentazione del Perception Package. I parametri configurabili dall'interfaccia sono i seguenti:

- range di intensità luminosa: varia da un minimo di 1550 Lux ad un massimo di 5000 Lux;
- range di rotazione sull'asse Y: copre tutti i valori tra 0° e 360°.

Inoltre, è possibile configurare la tipologia di distribuzione di probabilità dei valori casuale, in modo equivalente al Rotation Randomizer. La "Directional Light" presente all'interno della scena, per subire le modifiche attuate dal Light Randomizer, dovrà essere dotata dello script "Light Randomizer Tag", che dovrà essere aggiunto come suo componente.

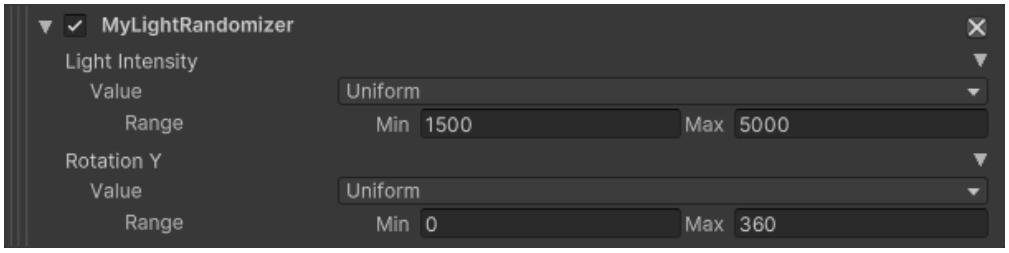


Figura 6.10: Interfaccia del Light Randomizer.

6.3 Gestione della simulazione fisica

La fase della simulazione fisica rappresenta un elemento di grande rilevanza all'interno del dataset generato per questo lavoro di tesi, in quanto costituisce la base per ottenere risultati orientati al realismo. Per implementare una simulazione fisica nel corso delle iterazioni del Simulation Scenario, è stato necessario utilizzare due

componenti fondamentali messi a disposizione da Unity: il Rigidbody ed il Mesh Collider.
Ogni prefab relativo agli oggetti presenti nel dataset è organizzato in una struttura gerarchica, come descritto precedentemente nel capitolo 5.3 in riferimento alla segmentazione delle parti degli oggetti. L'elemento principale corrisponde all'oggetto nella sua interezza e ad esso sono associati il componente Rigidbody ed il "Rotation Randomizer Tag". L'elemento subordinato, o gli elementi subordinati nel caso di un oggetto composto da più parti, fanno riferimento alla mesh dell'oggetto e includono i componenti Mesh Filter, Mesh Renderer, Mesh Collider e Labeling, quest'ultimo approfondito nel capitolo 6.1.3.

6.3.1 Componente Rigidbody

Il componente Rigidbody consente di influenzare il movimento e la posizione di un GameObject senza agire direttamente sulle sue proprietà di Transform (posizione, rotazione e scala), ma sfruttando l'applicazione di forze elaborate dal motore fisico di Unity. Un Rigidbody è affetto dalla forza di gravità che lo trascina verso il basso e, se opportunamente associato ad un componente di tipo Collider, reagisce alle collisioni con altri oggetti, simulando il comportamento fisico di un oggetto nel mondo reale. È possibile controllare un Rigidbody anche tramite script applicando forze di differenti tipologie tramite funzioni dedicate, come descritto nel capitolo 6.2.1 nella sezione "Gestione della sovrapposizione di oggetti".
Per accelerare il processo di creazione degli asset, è stato sviluppato uno script che consente di applicare un componente Rigidbody ad una selezione multipla di prefab, configurando i parametri su valori personalizzati. In particolare, il parametro "Collision Detection" è stato impostato su "Continuos", il che garantisce un calcolo preciso delle collisioni tra il collider associato al Rigidbody e altri collider presenti nella scena, specialmente in situazioni di movimento rapido.
Il parametro "Angular Drag" è stato configurato con un valore fisso di 15, che rappresenta un valore relativamente elevato. È stato scelto questo valore in quanto si è osservato che la rotazione imposta dal Rotation Randomizer poteva causare una rotazione inerziale significativa. Impostando un valore elevato per il parametro "Angular Drag" si simula una notevole resistenza dell'aria, il che limita la propagazione del movimento di rotazione e favorisce un comportamento più realistico degli oggetti.
Il parametro "Mass", corrispondente alla massa espressa in chilogrammi, non è stato gestito seguendo un approccio basato su valori realistici. Invece, è stato impostato in modo da rendere più realistica l'interazione tra oggetti di dimensioni diverse, assegnando un valore di massa proporzionale al volume dell'oggetto. A questo proposito è stato sviluppato uno script che ricava il volume dell'oggetto a partire dalle singole dimensioni dei collider associati al Rigidbody, per poi eseguire

un'interpolazione lineare, tramite la funzione `Mathf.Lerp`, generando un valore tra un minimo di 0.2 ed un massimo di 0.8, basandosi sul valore di volume ricevuto in input.
I restanti parametri offerti dal componente Rigidbody sono stati lasciati nella loro configurazione di default.
Il componente Rigidbody degli oggetti presenti nella scena viene manipolato tramite scirpt, oltre che durante la fase di gestione delle sovrapposizioni degli oggetti, anche precedentemente alla fase di cattura delle immagini. Per avere delle immagini consistenti e coerenti tra loro, è necessario che gli oggetti mantengano invariata la propria posizione nello spazio. Per questo motivo, prima che le immagini vengano catturate, le proprietà `velocity` e `angularVelocity` sono settate ad un valore pari a zero, in modo che gli oggetti non siano più affetti dalla simulazione fisica e rimangano statici.

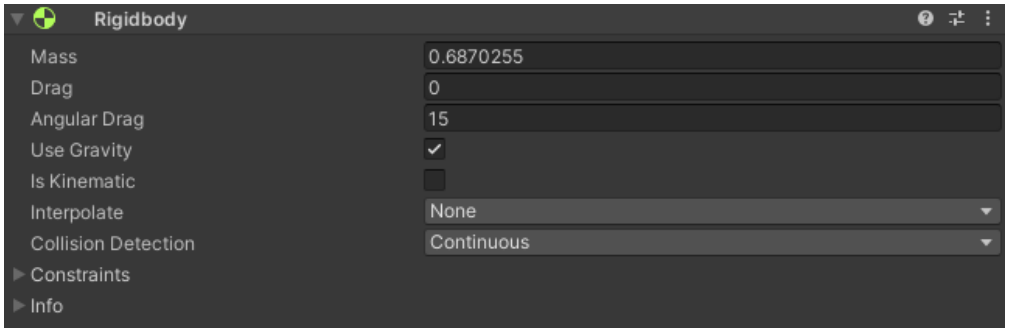


Figura 6.11: Interfaccia del componente Rigidbody.

6.3.2 Componente Mesh Collider

L'importazione di un modello 3D in formato OBJ all'interno di Unity genera, oltre al modello, un asset di tipo *mesh*. Il componente Mesh Collider ha lo scopo principale di creare un *collider* basato sulla *mesh* di un oggetto. Tra i vari componenti di tipo Collider forniti da Unity, è stato scelto il Mesh Collider in quanto, considerando le caratteristiche degli oggetti appartenenti al dataset, il calcolo delle collisioni effettuato basandosi sulle *mesh* è il più accurato. È inoltre possibile specificare se la *mesh* abbia una forma convessa, consentendo l'inclusione di parti concave dell'oggetto durante il processo di calcolo delle collisioni.
Al fine di garantire il corretto funzionamento del Mesh Collider, è necessario che l'asset relativo alla *mesh* dell'oggetto venga trascinato o selezionato nel parametro "Mesh", come mostrato in figura 6.13. Per velocizzare ed ottimizzare questa procedura, è stato implementato all'interno dello script precedentemente descritto nel

capitolo 6.3.1 un processo atto ad associare automaticamente a ciascun componente Mesh Collider la *mesh* corrispondente.

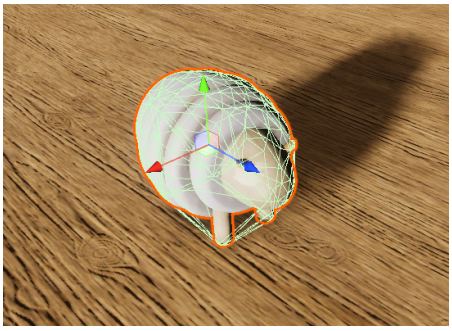


Figura 6.12: Visuale del componente Mesh Collider applicato ad un oggetto all'interno di una scena di Unity.

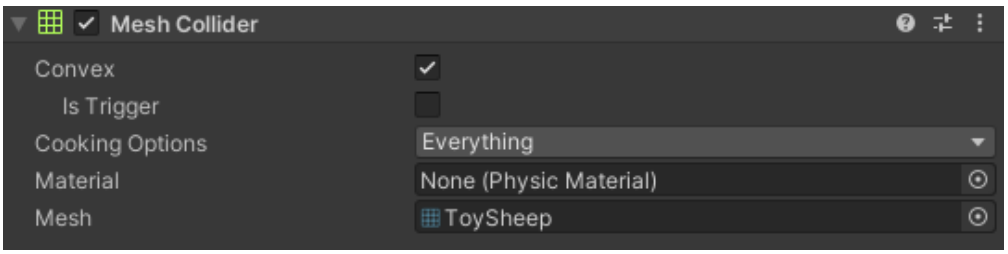


Figura 6.13: Interfaccia del componente Mesh Collider.

6.4 Meccanismo di cattura delle immagini

La Perception Camera è il componente del Perception Package responsabile della cattura delle immagini durante la generazione del dataset, come descritto precedentemente nel capitolo 6.1.2.
In accordo alle specifiche costruttive del dataset, è necessario che per ogni iterazione siano catturate 5 immagini, provenienti da differenti punti di vista posizionati su di una traiettoria ellittica. Per aumentare la complessità e la varietà del dataset, i punti di cattura non saranno fissi per ciascuna iterazione, ma saranno generati in maniera pseudo-casuale.
L'implementazione di questa caratteristica è stata realizzata mediante uno script dedicato, nominato Capture Position. Questo script viene aggiunto come componente alla camera virtuale di Unity che presenta già il componente Perception Camera. I parametri di questo script, modificabili dallo Unity Editor, sono mostrati in figura 6.14.

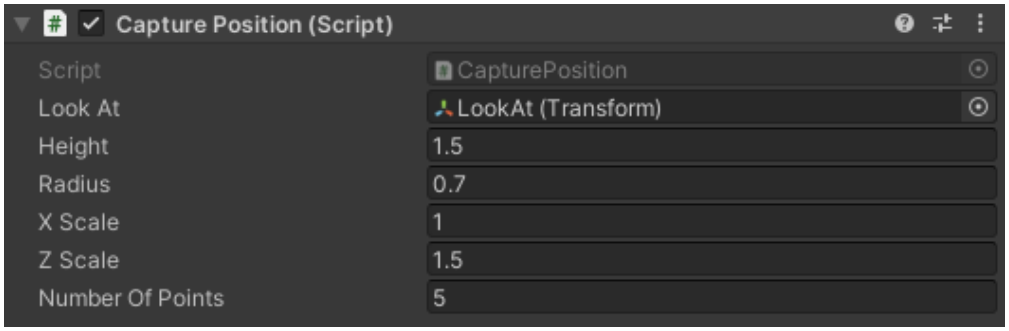


Figura 6.14: Interfaccia del componente Capture Position.

Il parametro "Look At" accetta come argomento una variabile di tipo `Transform`, la quale fa riferimento alle coordinate spaziali di un Game Object di tipo Empty posizionato all'interno della scena. Durante la cattura delle immagini la direzione della camera virtuale sarà sempre rivolta verso il "Look At". Questo Game Object, che ha la funzione di definire l'angolazione con cui vengono catturate le immagini, è posizionato con coordinate X = 0 e Z = 0, mentre la coordinata Y viene regolata in base al punto di vista che si vuole ottenere. I parametri successivi, ovvero "Height", "Radius", "X Scale" e "Z Scale", fanno riferimento alle caratteristiche geometriche e spaziali dell'ellisse che definisce la traiettoria della camera virtuale durante la fase di cattura. Mantenendo i parametri "X Scale" e "Z scale" uguali a 1, si otterrà una traiettoria perfettamente circolare, mentre valori diversi porteranno la traiettoria ad essere più o meno ellittica. Questi parametri devono essere regolati tenendo conto dell'area occupata dagli oggetti disposti sul piano di appoggio, in modo da avere un punto di vista il più ravvicinato possibile senza escludere degli oggetti, o porzioni di essi, dalle immagini catturate.
L'ultimo parametro, "Number Of Points", corrisponde al numero di immagini catturate per ogni iterazione ed è stato mantenuto pari a 5 durante tutto l'arco di generazione del dataset.
La funzione principale dello script, denominata `GeneratePoints`, restituisce una lista contenente elementi di tipo `Vector3`, ciascuno dei quali corrisponde alle coordinate spaziali dei punti in cui devono essere effettuate le catture delle immagini. Il campionamento di punti casuali lungo l'intera traiettoria comporterebbe la generazione di valori non uniformi, il che non garantirebbe una copertura ottimale delle diverse angolazioni da cui è possibile osservare la scena.
Il primo passo implementato per ottenere dei valori casuali ed unici per ogni iterazione, mantenendo al contempo l'uniformità lungo la traiettoria, è stato suddividere la conica in sezioni di uguale ampiezza. Successivamente, sono stati ottenuti gli angoli corrispondenti alle posizioni di cattura generando, per ciascuna sezione, un valore casuale compreso tra gli estremi della sezione corrispondente,

tramite la funzione `Random.Range`.
Le coordinate spaziali sono state ricavate mediante la creazione di variabili di tipo `Vector3`, alle quali sono stati assegnati parametri differenti per le tre coordinate X, Y e Z. La coordinata Y rimane costante per ogni posizione di cattura ed è definita dal parametro "Height", specificato tramite l'interfaccia dello script. Per determinare le coordinate X e Z, si è ricorso ad una formula che consente di ricavare la posizione di un punto appartenente ad un ellisse. Questa formula prende in considerazione le coordinate del centro dell'ellisse, il suo raggio e l'angolo corrispondente al punto di cattura. L'angolo è stato precedentemente convertito in radianti attraverso il prodotto con il termine `Mathf.Deg2Rad`. Le formule utilizzate per calcolare le coordinate X e Z sono le seguenti:

$$X = x_{center} + radius \cdot x_{scale} \cdot \cos angle$$

$$Z = z_{center} + radius \cdot z_{scale} \cdot \sin angle$$

Questo approccio garantisce che le 5 immagini catturate per ogni iterazione forniscano una visione completa della scena, offrendo punti di vista uniformemente distribuiti lungo la traiettoria ellittica.
L'effettiva cattura delle immagini avviene tramite la chiamata del metodo `RequestCapture` offerto dalla Perception Camera. Una volta che gli oggetti sono caduti sulla superficie d'appoggio e la fase di simulazione fisica è terminata, la camera virtuale viene posizionata nei punti generati dalla funzione `GeneratePoints`. Ad ogni riposizionamento viene invocata la funzione `RequestCapture`, in modo da effettuare la cattura dell'immagine RGB e delle ulteriori immagini specificate dai Labeler associati alla Perception Camera.
Questo approccio risulta inoltre essere ottimizzato in relazione alle tempistiche di generazione del dataset, in quanto sono sufficienti solamente 5 frame per effettuare le catture delle immagini.

6.5 Stampa del log di Unity su file esterno

Il processo di generazione del dataset implica un considerevole numero di iterazioni. Al fine di monitorare il progresso della generazione del dataset e, soprattutto, di identificare le cause di eventuali errori o anomalie verificatesi durante le simulazioni, è stato sviluppato un metodo per registrare informazioni relative all'andamento del processo su file di testo. Ogni file è relativo ad un singolo scenario, ed è salvato in una specifica locazione di memoria.
Per realizzare questa funzionalità, è stato creato uno scritp denominato Write Debug To File, associato al componente SimulationScenario. Questo script registra ogni informazione di Debug visualizzata all'interno della console di Unity su uno specifico file con estensione *.text*. Tale registrazione include sia le informazioni di

debug generate automaticamente sia quelle scritte manualmente tramite la funzione `Debug.Log(" ")`.
All'inizio del file sono riportate la data e l'ora di avvio della simulazione, seguite dall'indicazione dello scenario che verrà eseguito e dal numero di iterazioni programmate. Ad ogni iterazione viene registrato il conteggio incrementale delle iterazioni in corso, unito al numero di oggetti istanziati in quella specifica iterazione. Al termine dell'intera simulazione, nelle righe conclusive del file, vengono riportati il totale degli oggetti istanziati durante il processo e la data e l'ora di completamento della simulazione.

Capitolo 7

Conclusioni

7.1 Tempistiche di generazione e statistiche del dataset

Il dataset generato per questa tesi è stato realizzato tramite il software Unity, eseguito su un portatile Windows dotato delle seguenti specifiche tecniche:

- processore: AMD Ryzen 7 5800H;
- RAM: 32 GB DDR4 3200 Hz;
- scheda video: NVIDIA GeForce RTX 3060 Laptop GPU 6 GB.

Nella tabella 7.1 sono riportati i tempi impiegati per la generazione del dataset, suddivisi nei sei distinti scenari per ciascuna porzione di dataset.
Si può notare come lo scenario #05 abbia richiesto mediamente più tempo per essere generato. Tale disallineamento nelle tempistiche può essere attribuito al fatto che questo specifico scenario ha coinvolto un numero consistente di oggetti da istanziare, compreso tra 8 e 12, e l'ambiente virtuale, insieme alla sua illuminazione, è di natura più complessa rispetto allo scenario #04, in cui il numero di oggetti da istanziare è lo stesso.
Inoltre, è osservabile che il tempo medio necessario per completare un'iterazione, nei differenti scenari, rimane sostanzialmente simile all'interno delle diverse porzioni di dataset.
Per quanto riguarda la porzione di Test, si riscontrano tempi di generazione leggermente inferiori nelle porzioni "Similar" e "Unknown". Questo risultato può essere attribuito alla minore quantità di oggetti da gestire, rispettivamente 27 e 24 oggetti, in confronto ai 63 oggetti utilizzati nella porzione "Seen".
Il tempo totale impiegato per generare il dataset nella sua interezza equivale a 20 ore, 6 minuti e 8 secondi.

Dataset portion	Scenario	No. of iterations	Generation time	Average iteration time	Portion time
Train/Validation	#00	4000	02:08:19	00:01,9	14:37:21
	#01	4000	02:30:22	00:02,3	
	#02	4000	02:18:44	00:02,1	
	#03	4000	02:05:42	00:01,9	
	#04	4000	02:21:16	00:02,1	
	#05	4000	03:12:58	00:02,9	
Test - Seen	#00	600	00:17:31	00:01,8	2:05:24
	#01	600	00:20:43	00:02,1	
	#02	600	00:20:08	00:02,0	
	#03	600	00:18:51	00:01,9	
	#04	600	00:20:20	00:02,0	
	#05	600	00:27:51	00:02,8	
Test - Similar	#00	600	00:15:55	00:01,6	1:48:26
	#01	600	00:19:27	00:01,9	
	#02	600	00:17:03	00:01,7	
	#03	600	00:15:15	00:01,5	
	#04	600	00:17:06	00:01,7	
	#05	600	00:23:40	00:02,4	
Test - Unknown	#00	600	00:15:41	00:01,6	1:34:57
	#01	600	00:17:55	00:01,8	
	#02	600	00:15:36	00:01,6	
	#03	600	00:14:38	00:01,5	
	#04	600	00:15:50	00:01,6	
	#05	600	00:15:17	00:01,5	

Tabella 7.1: Tempo di generazione per le diverse porzioni del dataset.

Nella tabella 7.2 sono riportati il numero di oggetti istanziati per ogni iterazione, suddivisi per ciascuna porzione del dataset.
È possibile osservare che il numero medio di oggetti istanziati rimane pressoché costante per gli scenari che presentano lo stesso range di oggetti da istanziare, ovvero gli scenari #00 e #01, #02 e #03, #04 e #05.
Nel complesso, il totale di oggetti istanziati durante l'intero arco di generazione del dataset ammonta a 214.118.

Dataset portion	Scenario	No. of iterations	No. of istantiated objects	Average objects per iteration	Total objects per portion
Train/Validation	#00	4000	9711	2,43	147698
	#01	4000	10006	2,50	
	#02	4000	23997	6,00	
	#03	4000	23964	5,99	
	#04	4000	40008	10,00	
	#05	4000	40012	10,00	
Test - Seen	#00	600	1504	2,51	22200
	#01	600	1527	2,55	
	#02	600	3613	6,02	
	#03	600	3552	5,92	
	#04	600	5996	9,99	
	#05	600	6008	10,01	
Test - Similar	#00	600	1468	2,45	22142
	#01	600	1458	2,43	
	#02	600	3614	6,02	
	#03	600	3592	5,99	
	#04	600	6006	10,01	
	#05	600	6004	10,01	
Test - Unknown	#00	600	1477	2,46	22078
	#01	600	1471	2,45	
	#02	600	3596	5,99	
	#03	600	3611	6,02	
	#04	600	5967	9,95	
	#05	600	5956	9,93	

Tabella 7.2: Oggetti istanziati per le diverse porzioni del dataset.

Il dataset sviluppato per questa tesi include complessivamente 174.000 immagini, catturate all'interno di 34.800 scene distinte, ciascuna raffigurante una diversa disposizione di oggetti. La porzione di train/validation del dataset comprende 120.000 immagini catturate in 24.000 scene, mentre la porzione di test comprende 54.000 immagini catturate in 10.800 scene.

7.2 Sviluppi futuri

Con il dataset completamente generato si potrà procedere alla sua elaborazione, con l'obiettivo di ricavare le pose di *grasp* dalle scene in esso contenute. Per fare ciò si potranno sfruttare delle reti neurali, come Graspness Discovery [19], appositamente sviluppata per essere adottata all'interno di ambienti disordinati, o dei framework come TransGrasp [20], in grado di rilevare i contorni degli oggetti ed evitare le collisioni.
Sarà inoltre possibile ampliare e modificare il dataset, adattandolo ad eventuali esigenze future. Si potranno estendere gli oggetti attualmente disponibili con l'aggiunta di nuovi set di oggetti, modificare gli ambienti in cui gli oggetti vengono istanziati rendendoli più dettagliati e complessi, oppure stabilire delle nuove tipologie di immagini e raccolte di dati da generare in output.

Bibliografia

[1] Unity Technologies. *Unity Perception Package*. https://github.com/Unity-Technologies/com.unity.perception. 2020 (cit. alle pp. 1, 10).

[2] Hao-Shu Fang, Chenxi Wang, Minghao Gou e Cewu Lu. «GraspNet-1Billion: A Large-Scale Benchmark for General Object Grasping». In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Giu. 2020 (cit. alle pp. 2, 4, 5).

[3] Clemens Eppner, Arsalan Mousavian e Dieter Fox. «ACRONYM: A Large-Scale Grasp Dataset Based on Simulation». In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. 2021, pp. 6222–6227. DOI: 10.1109/ICRA48506.2021.9560844 (cit. alle pp. 2, 5, 6).

[4] Adithyavairavan Murali, Weiyu Liu, Kenneth Marino, Sonia Chernova e Abhinav Gupta. «Same Object, Different Grasps: Data and Semantic Knowledge for Task-Oriented Grasping». In: *CoRR* abs/2011.06431 (2020). arXiv: 2011.06431. URL: https://arxiv.org/abs/2011.06431 (cit. alle pp. 2, 6, 7).

[5] Hanbo Zhang, Jian Tang, Shiguang Sun e Xuguang Lan. *Robotic Grasping from Classical to Modern: A Survey*. 2022. arXiv: 2202.03631 [cs.RO] (cit. a p. 3).

[6] Jeannette Bohg, Antonio Morales, Tamim Asfour e Danica Kragic. «Data-Driven Grasp Synthesis—A Survey». In: *IEEE Transactions on Robotics* 30.2 (2014), pp. 289–309. DOI: 10.1109/TRO.2013.2289018 (cit. a p. 3).

[7] Rhys Newbury et al. «Deep Learning Approaches to Grasp Synthesis: A Review». In: *IEEE Transactions on Robotics* (2023), pp. 1–22. DOI: 10.1109/TRO.2023.3280597 (cit. a p. 3).

[8] Charles R. Qi, Hao Su, Kaichun Mo e Leonidas J. Guibas. «PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation». In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Lug. 2017 (cit. a p. 4).

[9] Berk Calli, Arjun Singh, James Bruce, Aaron Walsman, Kurt Konolige, Siddhartha Srinivasa, Pieter Abbeel e Aaron M Dollar. «Yale-CMU-Berkeley dataset for robotic manipulation research». In: *The International Journal of Robotics Research* 36.3 (2017), pp. 261–268. DOI: 10.1177/0278364917700714. eprint: https://doi.org/10.1177/0278364917700714. URL: https://doi.org/10.1177/0278364917700714 (cit. a p. 5).

[10] Jeffrey Mahler, Jacky Liang, Sherdil Niyaz, Michael Laskey, Richard Doan, Xinyu Liu, Juan Aparicio Ojea e Ken Goldberg. «Dex-net 2.0: Deep learning to plan robust grasps with synthetic point clouds and analytic grasp metrics». In: *arXiv preprint arXiv:1703.09312* (2017) (cit. a p. 5).

[11] Manolis Savva, Angel X. Chang e Pat Hanrahan. «Semantically-Enriched 3D Models for Common-Sense Knowledge». In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*. Giu. 2015 (cit. a p. 5).

[12] Miles Macklin, Matthias Müller, Nuttapong Chentanez e Tae-Yong Kim. «Unified Particle Physics for Real-Time Applications». In: *ACM Trans. Graph.* 33.4 (lug. 2014). ISSN: 0730-0301. DOI: 10.1145/2601097.2601152. URL: https://doi.org/10.1145/2601097.2601152 (cit. a p. 5).

[13] George A. Miller. «WordNet: A Lexical Database for English». In: *Commun. ACM* 38.11 (nov. 1995), pp. 39–41. ISSN: 0001-0782. DOI: 10.1145/219717.219748. URL: https://doi.org/10.1145/219717.219748 (cit. a p. 6).

[14] Steve Borkman et al. *Unity Perception: Generate Synthetic Data for Computer Vision*. 2021. arXiv: 2107.04259 [cs.CV] (cit. a p. 10).

[15] Unity Technologies. *Synthetic optimized labeled objects (solo) dataset schema: Perception package: 1.0.0-preview.1*. Visitato il 01/09/2023. Nov. 2022. URL: https://docs.unity3d.com/Packages/com.unity.perception@1.0/manual/Schema/SoloSchema.html (cit. a p. 17).

[16] Laura Downs, Anthony Francis, Nate Koenig, Brandon Kinman, Ryan Hickman, Krista Reymann, Thomas B. McHugh e Vincent Vanhoucke. *Google Scanned Objects: A High-Quality Dataset of 3D Scanned Household Items*. 2022. arXiv: 2204.11918 [cs.RO] (cit. a p. 27).

[17] Library of Congress. *Sustainability of digital formats: Planning for Library of Congress Collections*. Visitato il 11/09/2023. Gen. 2020. URL: https://www.loc.gov/preservation/digital/formats/fdd/fdd000507.shtml (cit. a p. 32).

[18] Unity Technologies. *The perception camera component: Perception package: 1.0.0-preview.1*. Visitato il 14/09/2023. Nov. 2022. URL: https://docs.unity3d.com/Packages/com.unity.perception@1.0/manual/PerceptionCamera.html (cit. a p. 40).

[19] Chenxi Wang, Hao-Shu Fang, Minghao Gou, Hongjie Fang, Jin Gao e Cewu Lu. «Graspness discovery in clutters for fast and accurate grasp detection». In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2021, pp. 15964–15973 (cit. a p. 58).

[20] Zhixuan Liu, Zibo Chen, Shangjin Xie e Wei–Shi Zheng. «TransGrasp: A Multi-Scale Hierarchical Point Transformer for 7-DoF Grasp Detection». In: *2022 International Conference on Robotics and Automation (ICRA)*. IEEE. 2022, pp. 1533–1539 (cit. a p. 58).